Probabilistically Time-Analyzable Complex Processor Designs

Mladen Slijepcevic

Computer Architecture Department

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

PhD in Computer Architecture

September, 2017

Probabilistically Time-Analyzable Complex Processor Designs

Mladen Slijepcevic

September 2017

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: Jaume Abella, PhD, Barcelona Supercomputing Center
Carles Hernández, PhD, Barcelona Supercomputing Center

Acknowledgements

I would like to express my deepest gratitude to my advisors, Francisco J. Cazorla, Jaume Abella and Carles Hernandez for their support, patience, help and time. I would like to thank to all members of the CAOS group at BSC for their help, Friday meetings and the discussions in the office we had. Specially the ones who were developing *SoCLiB* simulator at the same time as me. Working on the PROXIMA project was a very good experience for me, and I would like to thank to all members of the PROXIMA project for their collaboration. I would also like to thank Jan Andersson from Cobham Gaisler for hosting me during the 5-months Internship and for taking part in industrial project, as well as to all members of Cobham Gaisler hardware team.

On the personal side, I would like to thank to all my friends, ex/current PhD students and Serbian community in Barcelona, with whom life during my PhD studies was more enjoyable. This work and life in Barcelona would be impossible without the support and love of my parents and my sister.

This thesis has been financially supported by *Obra Social Fundación la Caixa* under the grant Doctorado "la Caixa" - Severo Ochoa, without which this PhD thesis and my life would be much different. This thesis has also received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the PROARTIS project (grant agreement 249100) and the PROXIMA project (grant agreement 611085), as well as the Spanish Ministry of Science and Innovation (grant TIN2012-34557 and TIN2015-65316-P). Collaboration at Cobham Gaisler was funded by HiPEAC.

Abstract

Industry developing Critical Real-Time Embedded Systems (CRTES), such as Aerospace, Space, Automotive and Railways, faces relentless demands for increased guaranteed processor performance to support new advanced functionalities without increasing the verification costs and the limited power budget. To cope with those needs, more complex processor designs are required. Unfortunately, CRTES have to go through a thorough functional and timing verification process. Functional correctness verification has to ensure that despite the presence of faults system's safety will not be compromised while timing verification focus on determining the worst-case execution time (WCET) of programs running in the processor. In CRTES it is of great importance deriving trustworthy and tight WCET estimates.

Current generation CRTES based on relatively simple single-core processors are already extremely difficult to verify and with the advent of multicore and manycore platforms this problem will be exacerbated. 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 and this makes problem of deriving safe WCET estimate worse. Therefore, timing analysis techniques need to include a lot of pessimism when using the advanced hardware features. Probabilistic Timing Analysis (PTA) has emerged recently as a powerful method to derive WCET estimates for critical tasks on relatively complex processors.

In this thesis we propose PTA-compliant hardware solutions to enable the use of more powerful and complex multicore and manycore processor architectures in future CRTES. Hardware solutions include arbitration policies and techniques to manage shared hardware resources (i.e. shared interconnection network and L2 cache), as well as approaches to obtain trustworthy WCET estimates on top of degraded hardware. PTA implemented on top of the proposed time-randomized designs allows modeling the timing behaviour of applications running in the processor in a probabilistical manner and therefore, WCET bounds can be made tighter.

Contents

List of Figures

Chapter 1

Introduction

1.1 Critical Real-Time Embedded Systems

Embedded systems are dedicated computing systems specially designed and used embedded within a larger system (typically mechanical or electronic). They are used in a large variety of domains such as bio-medicine, telecommunication, mobile, control systems. Since they are embedded, they have to satisfy limiting constraints such as size, weight, cost and power ones.

Real-time systems are a special type of embedded systems in which timing behaviour is as important as functional behaviour. They are deployed in cars, planes, trains, and unlike the conventional embedded systems, the software running in those systems demands for completing execution within specific time budgets. We can divide them according to their criticality. Criticality is the term correlated with the consequence of what would happen if the system fails and the notion of criticality is different in every domain and defined in different standards. For example, avionics DO-178B/C [RTCA and EUROCAE (2011)] standard defines 5 safety levels, from DAL-A to DAL-E, and each level imposes different certification steps. Missing a DAL-A safety goal would have catastrophic consequences, and may potentially cause a crash of the airplane. Missing a DAL-B one has a hazardous impact: it has a large negative impact on safety or performance. Conversely, missing DAL-E has no impact on safety, aircraft operation, or crew workload.

While in the past Critical Real-Time Embedded Systems (CRTES) followed a Federated Architecture approach [Obermaisser *et al.* (2009)], recently they have shifted to the Integrated Architecture paradigm. The former uses different hardware units for different functions and in that way physical separation allows timing and functional isolation. The latter, using a modular approach in which multiple functions are assigned to a single hardware unit, brings benefits in terms of reduced

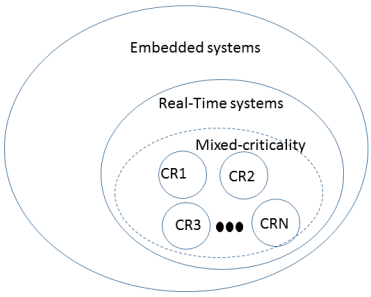


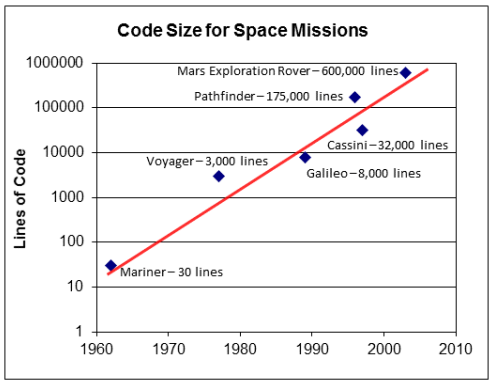
Figure 1.1: Critical Embedded Real-time systems

size, power and cost but also brings other issues such as more complex verification and certification[1] of the system.

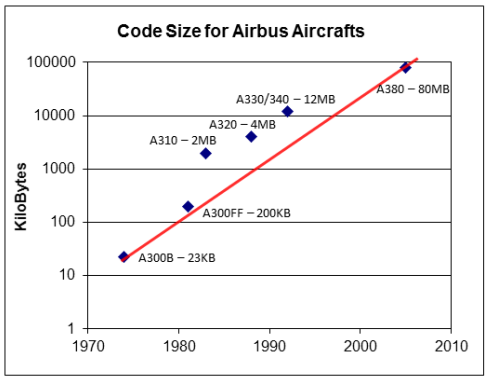
Mixed criticality systems are a special type of CRTES (see Figure 1.1) first introduced in [Vestal (2007)]. They integrate tasks with different criticalities (from most Critical CR1 to least critical CRN) on the same platform. A large software system typically consists of many software sub-systems executing concurrently and potentially cooperatively, but somehow independently from each other so that a failure in one sub-system must not negatively impact another. In particular, lower criticality systems must not negatively impact a higher criticality system. Enforcing this independence across systems in different criticality levels makes certification become more challenging.

Real-time industry has been growing across all markets [Clarke (2011)] and some sources suggest that this trend will continue in the future. CRTES are used not only to improve efficiency, performance, comfort and entertainment, but also to control safety-related functions in various market segments. Even though current systems use relatively simple hardware, every new generation deploys increasingly complex software as well as additional functionalities, which creates demand for more computational power. Safety-critical functions are, in many cases, complex thus requiring high computational power. Figure 1.2 shows the increasing code size in systems across avionics and space industries as a proxy of their increasing complexity. Particular examples for these and other domains are as follows:

[1]By term verification we mean the process of determining whether a system or component satisfies the conditions imposed at the start of the design phase. By term certification we mean that there is a written guarantee that a system or component is compliant with the specific (safety) standards and is acceptable for operational use.



(a) Space domain [D. Siewiorek (2006)]



(b) Avionics code [Edelin (2009)]

Figure 1.2: Size of code in different real-time domains

- **Automotive.** Electronics in automobiles provide an increasing amount of complex functionality, with features such as brake assist, active lane keeping, adaptive cruise control, etc. As a consequence of all those functionalities, a premium vehicle nowadays has more than 100 Million lines of code [J. Owens (2015)]. Also, some of the already existing safety-critical systems demand more computational power. For instance, in the early 90s the software for an ABS system required an ECU at 16MHz and only 128KB of memory. By 2004, it required an ECU at 250MHz (a 15X increase) and around 1MB of memory (an 8X increase) according to ARM data [Vittorelli (2004)].

- **Space.** Modern spacecraft have dedicated platform systems such as power management, communication, guidance and navigation, and payload systems, which in turn comprise systems such as infrared detectors, cryogenic systems, telescopes, etc. Space missions are becoming more autonomous and future applications will demand increased performance to provide computation-intensive on-board software [D. Siewiorek (2006)]. This is in-line with the current trend shown in Figure 1.2a.

- **Avionics.** A modern aircraft requires millions of lines of code just for its on-board control functions, such as guidance, navigation, anti-collision systems and other control algorithms as shown in Figure 1.2b.

Based on those trends, we can see that future CRTES will require more processing power and more complex hardware to accommodate the performance needs of the different applications that will use it. However, existing high-performance processors are known for their good average performance, but they pose a number

of difficulties to quantify their guaranteed performance, i.e. the maximum performance that can be guaranteed for a given application by the appropriate timing analysis technique.

1.2 Advanced hardware features and Timing Analysis techniques

In CRTES it is of great importance deriving trustworthy and tight Worst-Case Execution Time (WCET) estimates. WCET is important for verification and it is used for schedulability tests. For the schedulability, two properties of the WCET are desirable - timing isolation and composability. Timing isolation means that applications do not affect each other and timing composability means that the their WCET does not depend on the co-runners, so that the rest of the system does not have to be reanalyzed if we replace/modify one of the tasks in the system. These properties are critical for incremental verification of the system. However, those properties often lead to accounting for many pathological cases in high-performance hardware, which produces huge pessimism in terms of high WCET estimates. Timing analysis methods have difficulties to prove that certain high execution times cannot occur and, therefore, those scenarios need to be accounted for when estimating the WCET. In practice, those scenarios may not happen at all or may not happen simultaneously, so that the real WCET is far below the estimated one. Therefore, the complexity of the hardware platform, which delivers high average performance, is addressed with massive time over-provisioning. Hence, the only way to attain trustworthiness – the fact that the WCET estimate is above the real WCET – is achieved at the expense of renouncing to tightness, which is a resource waste and may even lead to scenarios where performance guarantees are insufficient (e.g. the braking system is guaranteed to brake in 2 seconds).

With increasingly complex CRTES, there are significant variations in execution time, and those variations are caused by the characteristics of the software in combination with the hardware platform on top of which software runs. Obtaining reliable (and as tight as possible) WCET estimates requires deep understanding of which of the low-level software and hardware features affect execution time, and how. On the software side, we can get different execution times depending on the input sets used, which would trigger different execution paths and loop boundaries, as well as the code and data memory layout. On the hardware side, features intended to increase average performance and throughput based on execution history and resource sharing, such as cache memories, branch prediction, speculation, multicores, potentially produce execution time variation.

There are many timing analysis techniques for estimating the WCET of tasks,

including static and measurement-based ones (and hybrid combinations thereof) [Wilhelm *et al.* (2008)]. We can classify them into two different paradigms: Deterministic Timing Analysis (DTA) and Probabilistic Timing Analysis (PTA).

Deterministic timing analysis techniques provide a single WCET estimate, and they are used on conventional (deterministic) hardware platforms. As mentioned before, they can be broadly classified in two complementary strands [Wilhelm *et al.* (2008)]: static timing analysis and measurement-based timing analysis. Static timing analysis builds upon the task code and an abstract model of the hardware. It needs to know all the details about the hardware and software and search for all possible states of the execution time. In the absence of that detailed knowledge, it needs to make pessimistic assumptions. These techniques find difficulties when hardware designs have complex features, whose timing behaviour depends on the execution history of the previous instructions in non-obvious ways, such as cache memories (e.g. a second level, L2, cache shared for code and data).

Measurement-based timing analysis estimates the WCET based on measurements obtained on the real hardware for some set of program inputs. On the highest-observed execution time (so called high-watermark) an engineering margin is added based on the experience of skilled users. This approach has worked well on simple hardware and has been successfully used in industry for many years. However, this approach has obvious limitations: one of the problems is the confidence that the user can have on whether the input data used to collect measurements triggers the WCET of the program on that hardware or it is reasonably close to it. Increasing input data coverage would increase the confidence, but covering all the input data range is impossible in practice. More advanced hardware features such as cache memories and multicores are also threats for the scalability of this approach due to the overwhelming difficulties to control the memory placement of data and code, and to control (at cycle level) how co-running tasks could interact in shared resources [Abella *et al.* (2015)]. Therefore, engineers face the complex challenge of having to size a reliable and tight engineering margin, which is a compromise between pessimistic overkill and the risk of underestimation. In general, there is not scientific evidence about the exact value to use.

Limitations of the deterministic timing analysis motivated some authors to propose a new timing analysis paradigm, called Probabilistic Timing Analysis (PTA) [Cazorla *et al.* (2013),Cucu-Grosjean *et al.* (2012)]. PTA provides WCET estimates in the form of distribution functions with an exceedance probability for each execution time value. Those probabilities can be chosen arbitrary low, based on the safety standards in the application domain (e.g. DO178B [RTCA and EUROCAE (2011)] for avionics) so that they are lower than the acceptable probability of failure in certified systems. PTA has also two variants, static and measurement-based one. In this thesis we focus on measurement-based probabilistic timing anal-

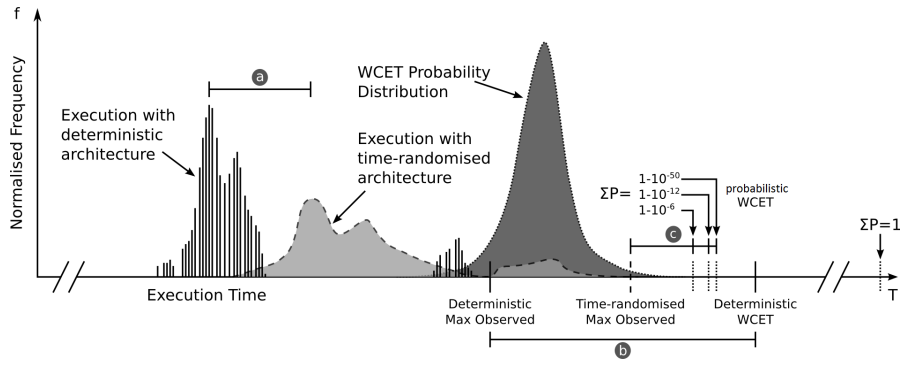


Figure 1.3: Execution time distributions. Taken from [Cazorla *et al.* (2013)]

ysis (MBPTA) since it is closer to industrial practice. MBPTA considers execution time measurements from observing end-to-end run of the program. Extreme Value Theory (EVT) uses those measurements as input [Cucu-Grosjean *et al.* (2012),Kotz & Nadarajah (2000)] to estimate the probabilistic WCET (pWCET), aka as exceedance function. The use of EVT requires that the observed end-to-end runs of the program are described with independent and identically distributed (i.i.d.) random variables, which can be obtained on time-randomized platforms, either by software or hardware means. The advantage of these platforms together with MBPTA is that they relieve the end user from having to control many low-level details to guarantee that WCET estimates obtained during the analysis phase are representative of the behaviour during operation. In Section 2.1 we provide more details on the Timing Analysis techniques and their requirements on the platform.

Figure 1.3 illustrates the relation between execution time profiles and WCET on probabilistic and deterministic architectures. In general, in time-randomized architectures execution time is shifted to the right (higher values) as indicated in the figure with mark (a). Usually execution times spread across a larger range and the distribution is smoother, with a long tail describing execution times which are less likely to happen.

In deterministic systems, execution times can have discontinuities due to the dependencies across events. Sometimes those dependencies can produce abrupt discontinuities in the execution time distribution. The "true WCET" lies somewhere above the maximum observed execution time obtained with measurement-based DTA. The difference between the maximum observed value and the true WCET is marked with the range (b) in Figure 1.3. When setting the WCET estimate, different scenarios can occur with static and measurement-based DTA. As explained before, the result of static analysis needs to account for some pessimism due to the (abundant) unknown information. Thus, the estimated WCET will be typically (far) above the true WCET. In the case of measurement-based DTA, the

WCET estimate is somewhere above the maximum observed execution time, but its location cannot be guaranteed to be above the true WCET.

PTA defines worst-case bounds with stated confidence levels, which can be chosen to match the degree of uncertainty present in the rest of the system being analyzed. The absolute maximum execution time produced by PTA on a time-randomized platform is many times greater than the WCET for DTA on a time-deterministic platform (see rightmost mark with accumulated probability 1), as it will correspond with the case where all instructions take the longest possible time to execute. However, since we are considering the execution time at which the cumulative probability exceeds the required level of confidence, this will allow a tighter WCET bound to be considered (c).

1.2.1 Multicores

The transition to multicore processors must enable higher level of guaranteed performance, together with reduction in energy, design complexity and procurement costs. However, the architecture of multicore processors poses several challenges on timing analysis. In multicores, sharing resources is necessary for efficiency reasons. However, the interference effects arising from the contention, arbitration and state perturbation of some resources requires much higher margins to be accounted for in the WCET analysis. Typical shared resources are the interconnection network, higher levels of cache memory and memory controllers.

Interference has been regarded as a key challenge [Cazorla *et al.* (2012)]. The purpose is deriving time-composable WCET estimates, aka WCET estimates that remain valid regardless of what tasks run in the other cores. From the timing perspective, the challenge with (mixed-criticality) real-time systems is in the need for solutions which can ensure temporal isolation between programs assigned to different criticality levels, so that their behaviour can be composable in the time dimension. If there is an absence of effective ways to deal with the pessimism of WCET analysis, then one may have to rethink the idea of combining more software on one system, which plays against resource efficiency.

The challenge of contention has been addressed from two different angles: (1) WCET analysis-centric and (2) architecture-centric. The objective of accounting for contention from a WCET analysis-centric approach is estimating how contention affects the timing behaviour of the task under analysis and derive stall times in the different resources against specific contenders. In general, this approach renders tight but fragile WCET bounds, since any variation in the timing behaviour of contenders changes stalls in arbitrary ways and defeats the whole analysis. Architecture-centric approaches, instead, focus on devising processor features and arbitration policies that help achieving time-composable behaviour that allows accounting easily and tightly for contention.

With the increasing number of cores on the chip, power dissipation on the chip is increasing, so the need of using smaller transistor technologies is necessary. However, its use is not straightforward since there is an increased number of permanent and transient faults. Those faults are usually not included in Timing Analysis techniques. During the testing period most of the permanent faults can be detected, but during operation hardware degradation can make latent errors grow enough to cause faults. Thus, accounting for the impact of faults and the corresponding countermeasures on WCET estimation is a need.

1.3 Contributions

This thesis proposes hardware designs to enable the use of more powerful and complex multicore and manycore processor architectures in future CRTES. It builds upon measurement-based probabilistic timing analysis (MBPTA) to deliver tight WCET estimates while preserving high average performance, thus increasing the level of guaranteed performance that a single system can provide.

Time composability is an important principle which is incorporated in every proposal. In that way, the WCET estimate of the task, calculated during the analysis phase, is not affected by any of the tasks running on the system at the same time.

The approach followed in this thesis is that interference can be treated in a probabilistical manner with support from time-randomized designs and therefore, WCET bounds can be made tighter by dismissing pathological scenarios that occur with too low probabilities. State-of-the-art multicores considered for use in CRTES, such as Infineon Aurix [Infineon (2012)], Cobham Gaisler LEON4 [Cobham Gaisler (2017)] and Freescale P4080 [FreeScale (2012)], consist of 3, 4 and 8 cores respectively. With the increasing number of cores, the degree contention on the shared hardware resources that timing analysis needs to account for in deriving reliable WCET estimates becomes substantial. The most relevant on-chip shared resources are (1) the interconnection network, (2) the shared L2 cache and (3) the memory controller.

In [Jalle *et al.* (2014)] authors have already analysed different MBPTA-compliant policies for shared buses and they provide solutions for small multicores (i.e. 4-core multicore), which are large enough for a first release of multicores in CRTES industry. Random-permutation arbitration policy proved being the most efficient one, but that solution was only evaluated and proven fair in the simple case where all requests have the same size. Whether such a policy is effective when requests have heterogeneous duration have not been proven yet. However, whenever performance needs are higher (i.e. future autonomous vehicles), on chip buses are inefficient providing sufficient bandwidth with an increasing number of cores (i.e.

in CRTES $\geq$ 8), so some other Network-On-Chip (NoC) topologies such as tree and mesh NoCs need to be used. Different NoC topologies produce high average performance for different needs (number of cores, types of communication, etc.), but the distributed nature of arbitration and the fact that links are shared, makes difficult deriving tight service time request bounds and therefore, WCET is often too pessimistic. Since MBPTA allows discarding contention scenarios that occur with negligible probability, making NoCs MBPTA-compliant is a promising approach.

The shared L2 cache poses significant challenge for CRTES since lines fetched by one core can be evicted by some other cores requests. To control inter-task interaction in the L2 cache, software cache partitioning [Liedtke *et al.* (1997),Mueller (1995),Kim *et al.* (2013),Ward *et al.* (2013)] and hardware cache partitioning [Paolieri *et al.* (2009a)] have been used so far. However, in the multicores this makes sharing pages or libraries hard, and poses some limitations on scheduling.

Finally, the use of smaller transistors helps providing more performance while maintaining low energy budgets; however, the use of nanoscale technology makes hardware fault rates increase noticeably which challenges time predictability and reliability. In particular, faults affect the temporal behaviour of the system in general, and WCET in particular. Timing analysis techniques can obtain WCET estimates that remains valid only under the same processor state on which measurements were taken (typically a fault-free state). To be able to provide reliable WCET estimates, it becomes mandatory for timing analysis tools to account for degraded hardware behaviour.

In this thesis we tackle those challenges by proposing hardware mechanisms, including arbitration policies and techniques to manage shared hardware resources (i.e. shared interconnection network and L2 cache), as well as approaches to obtain WCET estimates on top of degraded hardware. More concretely we propose the following:

- Multicore designs with increased number of cores and using NoC as an interconnection network:
  - We analyze and compare several MBPTA-compliant arbitration policies in multicores with buses and tree NoCs, in terms of homogeneous performance guarantees. We propose arbitration policies to meet the requirements of mixed-criticality systems.
  - We compare tree-based time-randomized and time-deterministic multicore designs in terms of WCET estimates.
  - We adapt wormhole-based mesh NoCs to be MBPTA-compliant and propose mechanisms for controlling the injection rate of the packets.

  - We propose a new credit-based control-flow mechanism for shared buses which is compatible with MBPTA and can significantly improve performance guarantees.

- We propose a new hardware mechanism to control inter-task interferences in shared time-randomised LLCs without the need of any hardware or software partitioning.

- We propose a holistic approach to deal with the timing impact of permanent and transient errors in future real-time systems implemented with smaller technology nodes.

The hardware designs and timing analyses proposed in this thesis are not applicable on conventional (time-deterministic) systems. Instead, to be able to apply MBPTA, end-to-end measurements needs to satisfy i.i.d. properties and processor resources build upon that requirement. However, hardware overheads to satisfy that requirement are affordable, and prototypes of those processors already exist [Hernández *et al.* (2015)].

1.4 Thesis Structure

The remaining of this Thesis is structured as follows:

- Chapter 2 covers background and state-of-the-art needed for this Thesis.

- Chapter 3 explains the methodology we use in this Thesis.

- Chapters 4, 5 and 6 cover MBPTA-compliant interconnection networks. More concretely:

  - Chapter 4 covers MBPTA-compliant tree-based multicores.
  - Chapter 5 proposes two techniques to enable use of MBPTA-compliant mesh NoC.
  - Chapter 6 proposes MBPTA-compliant arbitration policy that allows a fair sharing of hardware resources in bus-based multicores

- Chapter 7 proposes a technique to enable use of shared non-partitioned LLCs.

- Chapter 8 proposes a method which enables the use of hardware implemented with smaller transistors.

1.5 Publications

List of publications is

1. M. Slijepcevic, C. Hernández, J.Abella and F. J. Cazorla. "Boosting Guaranteed Performance in Wormhole NoCs with Probabilistic Timing Analysis" (Chapter 5), 20th Euromicro Conference on Digital System Design (DSD), August 2017

2. M. Slijepcevic, C. Hernández, J.Abella and F. J. Cazorla. "Design and implementation of a fair credit-based bandwidth sharing scheme for buses" (Chapter 6), Design, Automation and Test in Europe Conference and Exhibition (DATE), March 2017.

3. M. Slijepcevic, M. Fernandez, C. Hernández, J. Abella, E. Quiñones, F. J. Cazorla "pTNoC: Probabilistically Time-Analyzable Tree-Based NoC for Mixed-Criticality Systems" (Chapter 4) 19th Euromicro Conference on Digital System Design (DSD), August 2016

4. M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, F. J. Cazorla "Timing Verification of Fault-Tolerant Chips for Safety-Critical Applications in Harsh Environments" (Chapter 8) IEEE Micro, Special Series on Harsh Chips, November 2014

5. M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, F. J. Cazorla "Time-Analysable Non-Partitioned Shared Caches for Real-Time Multicore Systems" (Chapter 7) Design Automation Conference (DAC), June 2014

6. M. Slijepcevic, L. Kosmidis, J. Abella, E. Quiñones, F. J. Cazorla "DTM: Degraded Test Mode for Fault-Aware Probabilistic Timing Analysis" (Chapter 8) 25th IEEE Euromicro Conference on Real-Time Systems (ECRTS), July 2013

Other publications:

1. F. J. Cazorla, J. Abella, J. Andersson, T. Vardanega, F. Vatrinet, I. Bate, I. Broster, M. Azkarate-Askasua, F. Wartel, L. Cucu, F. Cros, G. Farrall, A. Gogonel, A. Gianarro, B. Triquet, C. Hernández, C. Lo, C. Maxim, D. Morales, E. Quiñones, E. Mezzetti, L. Kosmidis, I. Agirre, M. Fernandez, M. Slijepcevic, P. Conmy, W. Talaboulma "PROXIMA: Improving Measurement-Based Timing Analysis through Randomisation and Probabilistic Analysis" 19th Euromicro Conference on Digital System Design (DSD), August 2016

Chapter 2

Background

This chapter provides the background and state-of-the-art on timing analysis techniques used for WCET estimation, and on the existing hardware solutions to enable timing analysis of multicore designs.

2.1 Timing Analysis Techniques

Execution time of programs is difficult to predict because it depends on the software and hardware initial conditions and run-time environment. There are many factors, such as hardware features (e.g. initial cache state, or latencies of different hardware components depending on the input arguments), input vectors (e.g. data inputs which will determine the execution path of the program) or software features (RTOS interferences), which are important factors for execution time variation. The purpose of the Timing Analysis techniques in the context of real-time systems is estimating the Worst-case Execution Time (WCET), which is a proxy of the worst-case timing behaviour of the program under analysis. WCET estimates are later used as an input for the task scheduler, which is in charge of doing the schedulability analysis and, if possible, of scheduling tasks in a way that deadline misses are avoided.

As mentioned in Chapter 1 there are two main families of Timing Analysis techniques: Deterministic Timing Analysis (DTA) and Probabilistic Timing Analysis (PTA). Both of them have their Static, Measurement-based and hybrid variants. We note that Measurement-based variants are closer to industrial practice in many systems [Mezzetti & Vardanega (2011),Law & Bate (2016),Natale *et al.* (2016)].

2.1.1 Deterministic techniques

Deterministic timing analysis provides a single WCET estimate. Static Deterministic Timing Analysis (SDTA) uses as input an abstract representation of hardware and a structural model of the software. SDTA does not execute the code at all but, instead, builds strictly upon abstract interpretation. SDTA considers all possible inputs (values and states) of a program, combines control flow with the model of the hardware architecture and provides a sound bound (which cannot fail by definition) for this combination [Wilhelm *et al.* (2008)]. However, the necessary inputs are subject to inaccuracies and errors [Abella *et al.* (2015)], and incomplete and buggy documentation of hardware as well as unknown software parameters limit SDTA usability only to simple hardware and software whose documentation may be regarded as sufficiently reliable.

Measurement Based Deterministic timing Analysis (MBDTA) collects measurements on the real hardware platform with different input sets. It does need much less information than SDTA, which makes it more appealing for industry. However, some limitations of MBDTA make it hard to be used since the number and magnitude of uncertainties grows in pace with hardware and software complexity, so that using an engineering margin without scientific evidence to account for the unknown is increasingly risky. Among the limitations to apply MBDTA we find the following: (1) the user needs to provide inputs that lead to the actual WCET (or an execution time close to it) since he cannot measure all possible inputs, and (2) execution time measurements are collected on the test platform (software and hardware), which may not be identical to the one which will be deployed in the real system, which may produce some uncontrolled discrepancies. An example of MBDTA can be found in [Wenzel *et al.* (2008)].

Hybrid Deterministic Timing Analysis approaches [Wenzel (2006),Rapita Systems (2007)] improve MBDTA with some static information from the program control flow. For example, RapiTime [Rapita Systems (2007)] can collect measurements at fine granularity (e.g. basic block) and identify potential execution paths that have not been observed. Even though they improve MBDTA by reducing pressure on the user for producing input parameters which will lead to WCET, Hybrid approaches still lack *sufficient* confidence due to the unquantified uncertainty remaining.

2.1.2 Probabilistic techniques

Probabilistic Timing Analysis has emerged recently as an alternative to deterministic timing analysis techniques and it provides a distribution of WCET or probabilistic WCET (pWCET) curve. Each value of the pWCET curve, see a hypothetical example in Figure 2.1, has associated its residual risk – expressed as

a cutoff probability – which is the maximum probability with which one instance of the program will exceed that particular pWCET value.

Static Probabilistic Timing Analysis (SPTA) [Cazorla *et al.* (2013),Altmeyer & Davis (2014)], derives a probability distribution of the execution time for individual instructions or components statically from the model of the processor and software. Then, those distributions are combined with methods such as convolution to obtain the pWCET of the program under analysis. SPTA has been devised so far only for simple processors and faces similar problems with the timing model of the hardware and software inputs to those of SDTA.

In the case of Measurement-Based Probabilistic Timing Analysis (MBPTA) [Cucu-Grosjean *et al.* (2012)], the pWCET curve is computed based on the collection of end-to-end runs of the program under study on the target hardware. Given the N end-to-end runs, we could build the pWCET curve as the empirical complementary cumulative distribution function (ECCDF). The ECCDF is computed from the histogram of end-to-end runs. However, with N runs we could have derive pWCET estimates for probabilities down to $\frac{1}{N}$. Hence, for the a target exceedance probability of 10^{-12} we would need 10^{12} program runs, which is practically unaffordable.

To get higher precision we need to apply Extreme Value Theory (EVT) [Kotz & Nadarajah (2000)], a statistical method for approximating the tail of distributions on the collection of execution time measurements. Then the cutoff probability can be selected in accordance with safety standards in the application domain (e.g., DO-178B/C [RTCA and EUROCAE (2011)] for avionics). For instance, Figure 2.1 shows a pWCET curve in which a cutoff probability of 10^{-14} and the corresponding pWCET estimate are selected. The process of obtaining pWCET curve is explained later in Chapter 3.4. In any case, EVT provides a continuous function so that a pWCET can be obtained for any arbitrarily low exceedance probability.

Some prior work [Bernat *et al.* (2002)] introduces probabilistic hard real-time systems and the Execution Time Profile (ETP) at the granularity of basic blocks. Then, it applies convolution to the ETPs obtained from observations on the real platform. The main drawback of this approach is that the execution time is modeled directly by an empirical distribution function with no guarantees on whether observations upper bound what could happen once the system is deployed. Moreover, this method requires a very large number of samples to obtain a precise model. Later in [Hansen *et al.* (2009)] authors derive the WCET by applying EVT and fitting a Gumbel distribution to the block maxima of samples. However, it requires that the test inputs provided by the user provide sufficient coverage for a number of sources of execution time variation, which is hard to control by the end user.

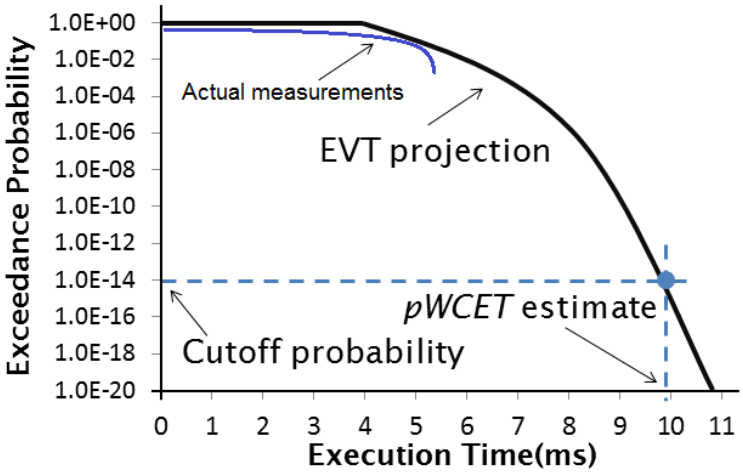


Figure 2.1: Example of the pWCET

A way to lower the needs on the high coverage needed and mitigate the impact of the sources of execution time variation by setting some constraints was proposed in [Cucu-Grosjean *et al.* (2012)]. That approach uses hardware features that break the dependence of the execution time on execution history, e.g. via randomization. This approach was proven to be successful in reducing the coverage needs [Francisco J. Cazorla & Abella (2013)]. It does not need to know memory addresses or value ranges for variables. Regarding path coverage, it can either be left on the user side [Milutinovic *et al.* (2017)] or be obtained automatically based on appropriate methods that exploit the advantages of time randomization [Kosmidis *et al.* (2014),Ziccardi *et al.* (2015)].

MBPTA has already been positively assessed for complex avionics case studies [Wartel *et al.* (2013),Wartel et al. (2015)] following a methodology close to industrial practice and MBPTA-compliant bus-based multicore designs have already been included in a FPGA implementation of the NGMP processor for the space domain [Hernández *et al.* (2015)].

Requirements of MBPTA

The use of EVT for WCET estimation purposes requires that its input, i.e. the collection of end-to-end execution times, can be described with independent and identically distributed random variables [Cucu-Grosjean *et al.* (2012)]. 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 [Feller (1966)]. Two random variables are said to be identically distributed if they belong to the same probability distribution [Feller (1966)]. Even though some

authors have proven that EVT can be applied even with less strict rules [Santinelli *et al.* (2014),Coles (2001)], in this Thesis we stick to the the original requirement, which allows an easier application of EVT.

Under MBPTA, response time of each execution component at a given granularity (e.g. an instruction) is attached with a distinct probability of occurrence. This attribute is described by a probabilistic Execution Time Profile (ETP), which is represented with the pair of vectors: $(\vec{l}, \vec{p}) = (\{l_1, l_2, ..., l_k\}, \{p_1, p_2, ..., p_k\})$. $\vec{l}$ is the vector which includes all its possible latencies, and p_i is the probability of the instruction taking latency l_i, accomplishing that $\sum_{i=1}^{k} p_i = 1$.

The existence of an ETP for each dynamic (executed) instruction enables the use of MBPTA [Abella *et al.* (2013)]: the fact the ETPs exist ensures that dynamic instructions behave as random variables and each potential execution time of the program has a distinct probability of occurrence .

Instructions may have *causal dependencies* among them [Abella *et al.* (2013)], such as data or control. Despite *causal dependencies*, independence and identical-distribution (i.i.d.) properties still hold across full program runs [Abella *et al.* (2013)].

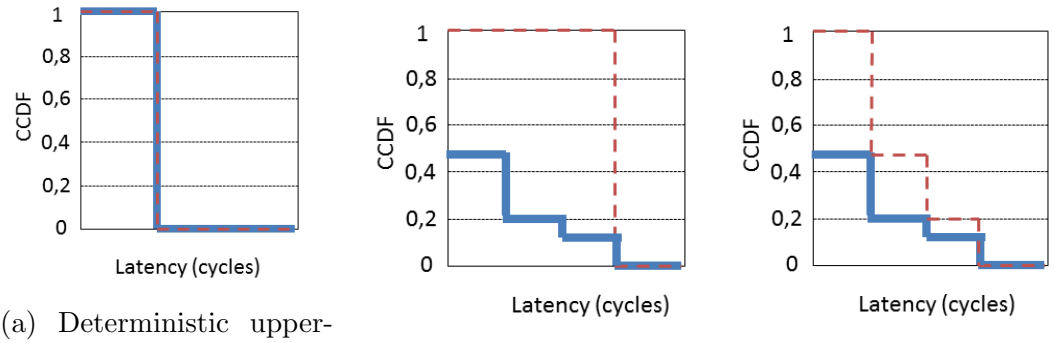
In a MBPTA-compliant processor two distinct modes are defined:

- *Analysis* mode is used during the collection of execution time measurements to estimate the WCET. The obtained WCET estimate must hold during operation. For that reason, the timing behaviour of the system as a whole, and therefore its individual resources in isolation, must be upper-bounded or exactly match that during operation. This guarantees that execution times experienced during operation will never exceed (probabilistically) those observed during the analysis phase.

- *Operation* mode is used once the system is deployed. During this mode some timing conditions enforced during analysis are unrestricted (or subject to fewer restrictions) and obtained execution times are lower or equal (probabilistically) than those during the analysis phase.

Since the timing of hardware resources has an impact on WCET analysis, during the analysis mode we need to upper bound their timing behaviour. There are two ways to perform such upper-bounding: (1) deterministically or (2) probabilistically.

Hardware resources can be divided into jitterless and jittery. Jitterless resources are those whose latency is constant (e.g. integer adder). Therefore, predicting their timing and accounting for their latency in any timing analysis technique is trivial. The ETP of those resources is $(\vec{l}, \vec{p}) = (\{L\}, \{1\})$, where L is the



(a) Deterministic upper-bound of a jitterless resource

(b) Deterministic upper-bound of a jittery resource

(c) Probabilistic upper-bounding

Figure 2.2: Types of upper-bounding

resource latency. In this case, at analysis mode, ETP_{bound} experienced during analysis matches $ETP_{resource}$ during operation. Thus, no modification is needed and we can upper bound its latency deterministically with its own ETP (the upper-bound is the red dotted line matching the ETP of the resource, Figure 2.2a).

On the other hand, jittery resources have variable latency which usually depends on input parameters (e.g. FP divider dividing by a power-of-2 versus by any other value) or execution history, or their combination (e.g. caches, pipelines, branch predictors). We could also upper bound their latency deterministically with $ETP_{bound} = (\{L_{max}\}, \{1\})$ where L_{max} is the highest latency, as shown on the Figure 2.2b. However, probabilistic upper-bounding is more efficient $ETP_{bound} = \left(\{l_1^{bound}, l_2^{bound}, ..., l_k^{bound}\}, \{p_1, p_2, ..., p_k\}\right)$ where for all latencies the exceedance probability of ETP_{bound} is higher or equal than that of $ETP_{resource}$. An example can be seen on Figure 2.2c. In general, probabilistic upper bounding allows for tighter WCET estimates than deterministic one.

2.2 High-Performance CRTES Hardware Platforms

In high-performance multicores we can find multi-level cache hierarchies as well as interconnection networks. Moreover, the requirement of higher performance a lower power leads to smaller technology nodes, which bring reliability issues. Next we provide background on those hardware designs in the context of MBPTA.

2.2.1 Cache memory

Cache memory is a hardware component hidden from the programmer and its efficient use relies on exploiting temporal and space locality of the memory accesses. It is a key component required for bringing high performance increase in high-end processors. From the perspective of timing analysis, there are two different types of caches: Time-Deterministic (TD) and Time-randomized (TR).

At a high level, in TD caches, placement and replacement policies build upon the data addresses and execution history to keep data in cache, since recently accessed addresses are typically the most likely to be accessed again in the near future and, therefore, they are more likely to be reused than other addresses, thus reducing execution time. However, such a strong dependence on the actual addresses accessed and the access patterns, as explained before, poses a problem for WCET estimation.

In TR caches data can be mapped to any set (randomly chosen on each run) and way (randomly chosen on every eviction) [Kosmidis *et al.* (2013a)], and thus, the dependence on execution history is significantly reduced, and hit and miss events have an associated probability for every cache access [Kosmidis *et al.* (2013b)].

Next we review the main placement and replacement policies used in TD and TR caches.

Time-Deterministic Caches

In Time-Deterministic caches locality has been exploited mostly by using modulo placement, and least recently used (LRU), First In first Out (FIFO) and Pseudo-LRU (PLRU) replacement policies. Most of the high-performance processors come equipped with two or even three levels of cache memory. Processors in the real-time domain have recently adopted these features. For instance, the NXP P4080 [FreeScale (2012)] or Cobham Gaisler Leon4 [Cobham Gaisler (2017)] have three and two cache levels respectively. For modulo placement, objects in consecutive positions in memory cannot conflict in cache. This is very good for performance as long as objects can be guaranteed to stay in specific (close) memory locations. This is often the case for many objects inside programs (e.g. neighbour variables, code inside a function), but for some others it is extremely difficult determining their memory location during unit analysis (aka before integration), since upon integration of different programs their relative location with respect some memory objects can change arbitrarily (e.g. with respect to shared libraries, RTOS services and data, etc.).

In terms of guaranteed performance, properly modeling cache behaviour is one of the main challenges for obtaining tight WCET estimates. Cache impact on WCET has been studied extensively in time-deterministic systems with a single

level hierarchy [Reineke *et al.* (2007)].

It is particularly challenging determining whether each particular cache access will hit or miss [Mueller (1994),Lesage *et al.* (2009),Hardy & Puaut (2008)]. This is affordable for the instruction cache [Mueller (1994)] and considering code strictly belonging to the task, but difficulties arise with the data cache since it is hard to statically determine memory addresses due to pointers, hidden array indexes, unknown stack alignment, etc. Using multilevel cache hierarchies further exacerbates this problem [Lesage *et al.* (2009),Hardy & Puaut (2008)].

In multicore systems, two approaches are often used to make cache behaviour more predictable: cache partitioning and cache locking. The first approach divides the cache into partitions and assigns different partitions to tasks or cores. In set-associative caches, partitions can be assigned as a certain exclusive number of sets [Chiou *et al.* (2000)] or ways [Mueller (1995)]. Both partitioning approaches can be achieved with either hardware or software means (compiler [Mueller (1995)] or OS support [Liedtke *et al.* (1997)]).

Cache locking [Campoy *et al.* (2001)] does not allow evicting locked lines until an unlock operation is done. It can be done through atomic instruction or defining the lock status of every cache way. While this technique is already used in single core processors to preserve cache contents upon the access to unknown memory addresses, it can also be used in multicore environments to limit inter-core interferences in shared caches.

Scratchpad memories [Banakar *et al.* (2002),Suhendra *et al.* (2005)] have been proposed as a more predictable hardware resource. Unlike cache memories, scratchpads are explicitly managed by the programmer and the memory blocks need to be moved from the main memory before their use. Therefore, their use is highly predictable. There is a similar objective between cache locking and scratchpad memories, that is, explicit controlling what is going to be cached. However, both mechanisms pose high responsibility on the user side, who must control what needs to be stored in cache, thus losing the advantage of cache memories, whose management is transparent for end users.

Time-Randomized Caches

In order to remove memory address dependence and mitigate history dependence, Time-Randomized multicore architectures rely on the use of cache memories implementing random placement and replacement policies [Mezzetti *et al.* (2015)]. Different cache layouts cause different cache conflicts among memory addresses, resulting in different execution times. Therefore, random policies are used to remove the dependence of execution time on the particular addresses accessed and to remove systematic pathological cases due to specific access interleaving. Thus, increasingly bad execution times occur with decreasing probability, leading

to pWCET estimates close to the average performance which, in turn, is close to that of the common case for TD caches.

Random placement [Kosmidis *et al.* (2013a)], which is based on a parametric hash function, for a given memory address and a random number called random index identifier (seed), provides a unique and constant cache set (mapping) for the address along the execution. Hence, during the execution of the program, the particular cache set for any given address is constant, but the placement is randomised across executions by modifying the seed of the used hash function. In this way, cache addresses are mapped to the same or different cache sets with particular probabilities, so that each memory access has hit/miss probabilities. Thus, the execution time of the program follows a distribution dictated by the random placement distribution, which leads to i.i.d. execution times, as needed for MBPTA.

Later, authors noticed that few lines can be (randomly) mapped to the same cache set with non-negligible probability, which increases WCET estimates despite the amount of data used by the program fits comfortably in cache. Hence, they proposed a design, random modulo [Hernández *et al.* (2016)], that employs a new random placement policy that retains most of the advantages of modulo placement by avoiding conflicts across contiguous addresses, while still meeting the requirements of MBPTA. In particular, random modulo performs a random permutation of the addresses within a memory page so that conflicts across lines in different pages are still random, but conflicts across lines in the same page cannot occur.

Random replacement [Kosmidis *et al.* (2013a)] ensures that every time a memory request misses in cache, a way in the corresponding cache set is randomly selected and evicted to make room for the new cache line. This ensures that (1) there is independence across evictions and (2) the probability of a cache line to be evicted is the same across evictions.

Moreover, in Time-randomized architectures, multi-level unified data and instruction caches [Kosmidis *et al.* (2013b)] can be efficiently analysed, including different write, write-allocation and inclusion policies among the different levels.

2.2.2 Interconnection Networks

Bus-based interconnect are used in multicores with a small number of cores. They have a centralized arbitration and it is easy to obtain tight bounds for their response time.

Paolieri *et al.* (2009a) have shown that in general, priorities cannot be used for arbitration if all cores need to run real-time tasks. Instead, policies such as round-robin and TDMA can ensure that all cores will be granted access to the shared resource eventually. If round-robin policy is used, then the delay every request can suffer can be upper bounded, as analysed in [Paolieri *et al.* (2009a)].

TDMA arbitration policy [Kelter *et al.* (2011),Schranzhofer *et al.* (2010)] fulfills the isolation property for the applications with different criticalities and therefore, it is suitable for CRTES. It applies time sharing between the requests of the different contenders. Time is typically split across cores homogeneously and it is also common using time slots whose duration matches the longest duration of any request Jalle *et al.* (2013a). Assuming that the duration of a request is unknown a priori (i.e. whether it will hit/miss in L2, whether it will produce a dirty line eviction in L2, etc.), TDMA will typically allow requests to be issued only in the first cycle of the corresponding slot for each core. Allowing a request whose duration is unknown to be issued at any other time could prevent requests from other cores being issued at their expected time, which is not allowed for being able to estimate the WCET. Among those two policies round-robin provides lower WCET and higher average performance with little burden for the end user in the context of SDTA and MBDTA [Jalle *et al.* (2013a)].

In MBPTA-compliant multicores we need to upper-bound the timing behaviour during the analysis phase, but also end-to-end execution times need to satisfy the i.i.d. requirement. These arbitration policies from time-deterministic bus-based systems have been shown to meet the requirements of MBPTA:

- **Round-robin (RR)**. In the context of MBPTA, when the time alignment of the requests w.r.t. the round-robin can be any – which is the least restrictive case from the user point of view – one needs to assume that each request will experience the maximum arbitration latency [Jalle *et al.* (2014)], which can be enforced at *analysis time* with the worst-case mode [Paolieri *et al.* (2009a)]. For the case of round-robin the highest latency a request can suffer due to the access to the bus is: $(N_c - 1) \times L$, where N_c is the number of cores (contenders) and L the bus latency. This corresponds to the case in which a request from one core has to wait for one request from each other contender to complete.

- **TDMA** This arbitration policy cannot be directly analysed by MBPTA because we cannot prove that the delay a request experienced during *analysis* time is an upper bound of the delay during *operation*. However, instead of augmenting every request delay like in the RR case, we only have to pad the end-to-end execution time with the following number of cycles: $N_c \times L - 1$, and then apply MBPTA. This approach allows tighter WCET estimates than RR [Panic *et al.* (2015)], although average performance is worse than for RR.

  While time-deterministic policies can be used in the context of MBPTA, some other policies have been specifically designed for MBPTA:

- **Lottery (LOT)**. The arbiter grants access to the different contenders randomly on each arbitration [Lahiri *et al.* (2001)]. To be MBPTA-compliant,

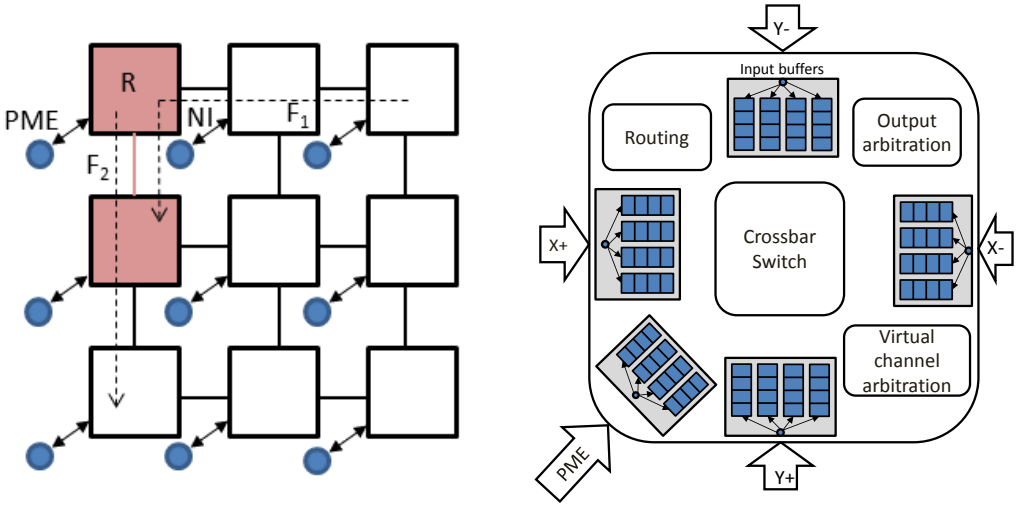
arbitration is always performed across all contenders (N_c) and only the one chosen can access the bus in that slot regardless of whether it has any pending request [Jalle *et al.* (2014)]. Therefore, large arbitration times occur with decreasing probabilities.

- **Random permutations (RP)**. Lottery arbitration may lead to theoretically infinite contention delays since there exists a non-null probability a contender waits long time to get granted access to the bus. With random-permutations [Jalle *et al.* (2014)] in each arbitration window, comprising one slot per contender, the order of the slots is randomly generated. For instance, with 3 contenders we could have the following arbitration windows (random permutations of 1,2,3): <2,1,3>, <1,2,3>, <3,1,2>, etc. This policy bounds the largest number of slots a contender may wait to get the bus (unlike lottery arbitration).

However, when the number of cores increases buses cannot provide sufficient bandwidth and NoCs are a better choice. A NoC is built with routers (R), links and network interfaces (NI). A Processing/Memory element (PME) is connected with a network interface to a router, and routers are connected via links among them and are in charge of forwarding the packets at the input ports to the appropriate output port. The PME can be either a processor core, main memory, I/O, etc. Connection of the routers defines the network topology. In this thesis we focus on two of the most used NoC topologies – tree and mesh topology.

Trees are used in real processors [Benini *et al.* (2012),Panic *et al.* (2014)] and they are suitable for $N - to - 1$ communication. Their routers are simple and typically perform 2-to-1 multiplex arbitration. However, since many flows can interfere, deriving tight WCET estimates is a challenge.

Routers for mesh topology are more complex, but the mesh is more scalable topology. We can see an example in Figure 2.3a. Network routing defines the way messages will traverse the NoC from the source to the destination node. The most used routing scheme is XY routing. That means that each router will have 5 ports $(X^+, X^-, Y^+, Y^-, PME)$ In the network, several traffic flows may be active at the same time and the routers and links are shared between different communication flows (e.g. flows F1 and F2 share two routers and one link between them in the figure) so usually timing analysis techniques need to account for very pessimistic cases that may occur systematically. If we take a closer look to one router in Figure 2.3b, we can see that the problem of interference arises since the flows can be sharing Input buffer allocation, Crossbar Switch, and output arbitration stages of the router.



(a) mesh topology (b) Closer look to the mesh router

Figure 2.3: Example of the mesh NoC

2.2.3 Reliability

The use of nanotechnologies increases both, permanent faults due to fabrication defects as well as transient fault susceptibility due to the low charge required to alter transistors state.

Permanent faults. Process variations [Bowman *et al.* (2002)] in CMOS technology affect mostly device timing and power profile, making them behave differently from expected. The relative impact of process variations becomes more significant for small devices, which specially jeopardizes cache memory bitcells operation, since they are implemented using the smallest features allowed. Moreover, the random component of process variations cannot compensate across delay paths given that bitcells are typically implemented with few transistors [DeMicheli *et al.* (2009)]. Hence, in general cache memories become the reliability bottleneck when scaling CMOS technology due to process variations.

Small defects unable to produce faults when processors are deployed, become large enough during operation due to degradation. The number of such defects and their probability to become actual faults during operation increases with technology scaling as shown in [Guertin & White (2010)]. This imposes serious challenges in the use of small technology nodes in CRTES, since CRTES expected lifetime is longer than that of its hardware components. For instance, it has been predicted that Failures in Time (FIT)[1] rates for 65nm technology nodes lead to lifetimes below 10 years [Guertin & White (2010)], which is already too short for the space

[1] 1 FIT corresponds to 1 failure per 10^9 hours of operation.

and avionics domains where systems may last for more than 20 years [Guertin & White (2010)]. Technology nodes below 65nm further reduce lifetime below 10 years, thus affecting other industries such as the automotive one.

Transient faults. Technology scaling also decreases the charge required to alter the proper behavior of transistors, which is known as *Qcrit*. By decreasing Qcrit, transient faults due to cosmic rays or alpha particles whose charge was below Qcrit for some technology nodes, can jeopardize correct operation when scaling technology down because their charge becomes relatively higher [Massengill *et al.* (2012)]. As a result, technology scaling increases soft error rates due to transient faults.

In order to deal with errors during operation, means for error Detection, Correction, Diagnosis and Reconfiguration (DCDR) are set up [Gizopoulos *et al.* (2011)] and hardware is allowed to operate on a degraded mode due to permanent faults. While those methods have been devised to guarantee functional correctness, they may often produce an inordinate impact in performance, thus invalidating WCET estimates of tasks computed for a non-degraded system. Little work has been done to attain time-predictability in the presence of degraded (non time-randomized) caches used together with STA but still neglecting the impact of error DCDR [Abella *et al.* (2011c),Hardy & Puaut (2013)]. However, methods considering the impact in WCET of all fault-tolerance features have not been yet devised.

Chapter 3

Methodology

The methodology used in this thesis consists in collecting end-to-end execution times of the tasks running on the processor in order to obtain 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: simulators and platforms we used, benchmark suite, and the process to obtain WCET estimates.

3.1 Simulators

In this thesis, for modeling the processor setup, we have used a simulator based on the SoCLib simulation framework [Pouillon *et al.* (2009)]. SoCLib is a cycle-accurate SystemC simulator that can be used for virtual prototyping at microarchitecture level. The purpose of the cycle-accurate simulator is to obtain all quantitative metrics (i.e. execution time, cache miss rates, etc.) with fine granularity.

This simulator has separated functional and timing components, which means that it creates two different design spaces: (1) the *functional emulator* and (2) the *timing simulator*. The (1) functional emulator executes instructions of a particular Instruction Set Architecture (ISA) and provides all the information about instructions like the instruction address, registers, instruction type, operation results and memory address in case of a memory operation. The (2) timing simulator models the timing behaviour of the instructions in a particular implementation of the ISA, e.g. it simulates the different timing behaviour in case of a cache hit or a miss and eventually the delay introduced by the access to a higher level cache or the memory, pipeline stalls, etc. This approach brings high flexibility since the functional part can be easily changed to emulate different ISA. We have used an emulator for a PowerPC 750 processor core [Freescale (1997)] due to the recent interest of Airbus in e500 core present in the MPC85xx and the new eight-core,

Table 3.1: Different configuration of shared resources

Resource	*Configurations*
L2 cache	Partitioned/non-partitioned
Interconnect (number of cores)	bus (4-8) /Tree NoC (up to 16)/mesh NoC (up to 36)
Memory controller	Fixed-latency/random-permutation

the P4080 [Cazorla *et al.* (2010)].

SoCLib simulator has integrated only a bus as interconnection network. In order to simulate the timing behaviour of all transactions in the NoC, we decided to model the NoC behavior using the gNoCsim [NanoC (2010)] simulator. The gNoCsim simulator is an event cycle-accurate and flit-accurate simulator developed in C/C++. This simulator was developed in the context of the NaNoC project to explore different topologies and routing algorithms and has been extensively used during the last years by both academics [Flich & Bertozzi (2010),Gorgues *et al.* (2014)] and practitioners [Dubois *et al.* (2011)]. The gNoCsim simulator allows two different operation modes: stand-alone with synthetic traffic patterns and slave-mode in which the simulator is coupled with another simulation tool, SoClib, in our case.

3.2 General architecture

The architecture of the multicore processor considered in this thesis is shown in Figure 3.1 and different configurations of the shared hardware resources are listed in Table 3.1.

Regarding core design, all cores are homogeneous, and we consider a design similar to those that have been successfully analyzed with PTA techniques so far. Its design can be seen in Figure 3.1.

The core architecture consists of a pipeline where instructions are fetched (F), decoded (D), executed (E) and results written back (WB) in order. First level instruction (IL1) and data (DL1) caches are private per core. The operations occurring in each stage are as follows:

- Fetch (F) stage. The instruction translation look-aside buffer (ITLB) and the IL1 are accessed in parallel to obtain the next instruction to be executed. Instructions fetched will be placed in a queue between F and D stages.
- Decode (D) stage. Instructions are decoded. This stage is, in essence, an extra delay in the pipeline.
- Execute (E) stage. Non-memory instructions are executed with a fixed latency that depends solely on the type of operation. For instance, an integer

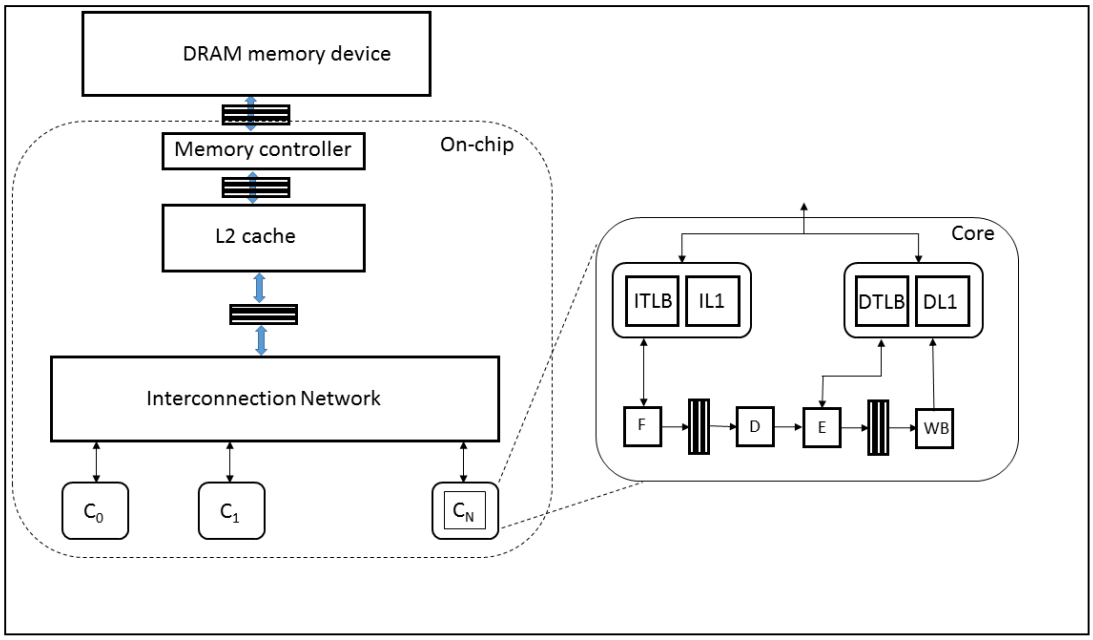


Figure 3.1: Schematic of the processor architecture.

addition will take always 1 cycle. Load instructions access the DL1 and DTLB in parallel. Indeed, they also access the write buffer to ensure data consistency. Store operations compute its address and get the data to be stored, but do not proceed to cache. Results are placed again in a queue and written back to the register file or to DL1 during the WB stage, when exceptions have been checked.

- Write-back (WB) stage. Results (if any) from all operations but stores are written into the register file. Store operations typically write their results in a store buffer to be transferred to cache as needed without blocking the pipeline. Only if the store buffer is full the pipeline is blocked.

DL1 and IL1 misses are propagated to the L2 cache through the shared communication network. If the DL1 is write-through all store operations are also forwarded to the L2. Alternatively, if it is write-back, only dirty cache lines are sent to the L2 on an eviction. Eventually, if the L2 is inclusive with DL1 it may raise evictions to the DL1 cache on a L2 eviction. Cache coherence is provided by software means and hardware support is only in place to guarantee L2 inclusivity with DL1.

Whether we consider write-through or write-back caches, as well as the particular cache parameters (size, associativity and bytes/line) is described in the

corresponding chapters since configurations stressing different components may differ.

3.3 Benchmarks

We use the EEMBC [Poovey *et al.* (2009)] automotive benchmark suite, which has been developed by the Embedded Microprocessor Benchmark Consortium. It is designed to analyze, test, and improve multicore processors in automotive, industrial, and general-purpose applications. All benchmarks consist of a main loop with few function calls in its body. The input data are embedded in the application, which emulates memory mapped input from the sensors, and in that way each iteration of the loop will use a different set of values. The list of EEMBC benchmarks used is detailed in Table 3.2.

Benchmarks are run without real-time operating system (RTOS) support, in order to assess the performance of hardware designs. On the one hand, this required some software hard-coding to allow benchmarks run directly on hardware, and on the other hand, enabled the evaluation of non-communicating tasks statically allocated to cores and running end-to-end, so not allowing any preemption, migration or synchronisation across tasks. We regard the evaluation of those other activities as mostly related to the RTOS, which is beyond the scope of this thesis.

Table 3.2: Benchmarks used in our simulation environment

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

3.4 MBPTA process

The outcome of the MBPTA process is a pWCET curve such as the one in Figure 2.1. MBPTA consists of the following five steps [Cucu-Grosjean *et al.* (2012)]:

1. **Collecting observations:** The first step is collecting a given number of end-to-end execution times of the program under analysis. Typically this number is $N = 1000$. In case that we need more measurements (explained in the following steps), additional $N_{delta} = 50$ measurements are collected and included in the sample. At every collection round we check if data satisfy the i.i.d. properties. For that purpose we need to apply (1) the two-sample Kolmogorov-Smirnov (KS) test [Feller (1966)], for checking identical distribution and (2) the runs-test [Bradley (1968)] to check fulfillment of the independence property. Both tests use a significance level of $\alpha = 0.05$, thus meaning that under normal conditions 5% of the tests will be failed. On a test failure we increase the number of measurements until the tests are passed. The fact that execution time is probabilistically i.i.d. in our architectures by construction guarantees that the execution time sample will eventually pass the statistical i.i.d. tests.

2. **Grouping:** In this step we use the Block Maxima method: we divide the sample into blocks of a given size (typically 25 or 50) and pick up the maximum value of each blocks. While there is not a strict rule for the selection of the block size, larger blocks allow discarding values not belonging to the (high) tail, but they must not be too large so that the population of maxima becomes too small. Thus, we use block sizes comparable to those in [Cucu-Grosjean *et al.* (2012)].

3. **Fitting:** An EVT distribution needs to be fit to the (maxima) execution times obtained in the previous step. The General Extreme Value (GEV) distribution, denoted as F, has as general expression:

$$F_{\xi}(x) = \begin{cases} e^{-(1+\xi\frac{x-\mu}{\sigma})^{\frac{1}{\xi}}} & \text{if } \xi \neq 0 \\ e^{-e^{-\frac{x-\mu}{\sigma}}} & \text{if } \xi = 0 \end{cases}$$

where ξ is the shape parameter, σ defines the scale and μ location parameter. For different values of ξ we have three different EVT distribution families: Gumbel ($\xi = 0$), Frechet ($\xi > 0$) or Weibull ($\xi < 0$). It has been proven in [Cucu-Grosjean *et al.* (2012)] that the Gumbel distribution fits well the problem of WCET estimation. In our process we use the exponential tail (ET) test [Garrido & Diebolt (2000)] to validate that a Gumbel distribution maxima execution times.

4. **Convergence:** In this step we determine if collecting more execution time measurements is needed, or whether adding more measurements would change the obtained WCET estimate. To do so, at every round we compare it to the previous one (except for the first) and we calculate the continuous rank probability score (CRPS) metric, which is defined as $\sum_{i=0}^{\infty}[f_X(i) - f_Y(i)]^2$ where $f_X(i)$ and $f_Y(i)$ represent distribution functions from the current and previous round. If CRPS is below a given threshold (e.g. 0.1) we do not need to collect more measurements. If not, we collect N_{delta} additional execution time measurements. It is worth mentioning that, since the execution times of finite programs can be upper bounded with a Gumbel distribution (so with an exponential tail), the ET test will be eventually passed and the distribution will converge to a Gumbel distribution by construction [Cucu-Grosjean *et al.* (2012)].

5. **Tail extension:** The resulting Gumbel EVT distribution is used to compute the pWCET estimate for the exceedence probability of interest. In our experiments we usually take the value 10^{-13} per run, in line with random hardware failure probabilities per hour allowed for the highest criticality levels in CRTES.

Chapter 4

Tree-based PTA-compliant NoCs

The use of networks-on-chip (NoC) in real-time safety-critical multicore systems challenges deriving tight worst-case execution time (WCET) estimates. This is due to the complexities in tightly upper-bounding the contention in the access to the NoC among running tasks. So far PTA has only been tested on small multicores comprising an on-chip bus as communication means, which intrinsically does not scale to high core counts. In this Chapter we propose *pTNoC*, a new tree-based NoC design compatible with PTA requirements and delivering scalability towards medium/large core counts, suitable for single and mixed-criticality setups. Moreover in this Chapter we compare time-deterministic and time-randomised multicore designs in terms of WCET estimates.

4.1 Introduction

Among multicore shared resources the network-on-chip (NoC) has prominent impact on programs' execution time and pWCET, as it connects cores to memory and/or shared cache levels. In the context of non time-randomized multicore architectures, also known as time-deterministic architectures, several NoC designs have been evaluated including meshes [Schoeberl *et al.* (2012)], rings [Panic *et al.* (2013)] and buses [Jalle *et al.* (2014)]. Among existing NoC designs, only buses have been proven MBPTA-compliant [Jalle *et al.* (2014)] for different arbitration policies. However, bus scalability is limited since its latency increases rapidly with the number of cores [Roca *et al.* (2012)]. Further, in the context of MBPTA, proposed arbitration policies offer homogeneous guarantees and performance across cores, which do not match the heterogeneous bandwidth requirements in future mixed-criticality multicore real-time systems [Vestal (2007)].

In this Chapter we propose *pTNoC*, a new tree NoC design that overcomes the limitations of homogeneously-arbitrated buses to manage the abundant traffic

between cores and memory and/or shared caches. Trees are chosen since they can scale to higher-core counts, have been shown to work with time-deterministic architectures [Panic *et al.* (2014)] and are implemented in real processors such as the P2012 [Benini *et al.* (2012)]. The challenge lies on making a tree NoC MBPTA-compliant while providing high average performance and heterogeneous – configurable – guaranteed bandwidth allocations under a low complexity and energy envelop. In particular, this Chapter makes the following contributions:

1. We analyze several MBPTA-compliant arbitration policies for tree NoCs, and show that trees scale better to high core counts than buses in terms of homogeneous performance guarantees.

2. We propose arbitration policies to enable fine-grain and flexible heterogeneous bandwidth assignments to better match the requirements of mixed-criticality systems. This is implemented through small changes in the router arbitration policy.

3. We provide a complete evaluation in terms of WCET estimates, average performance, power and area, and compare our *pTNoC* against the only MBPTA-compliant NoC so far, a shared bus [Jalle *et al.* (2014)], showing that our *pTNoC* outperforms the MBPTA-compliant bus in all metrics.

4. We compare a tree-based time-deterministic multicore and our tree-based time-randomised multicore in terms of WCET estimates.

4.2 Reference Multicore

As explained before in Chapter 3, we consider a processor and memory architecture in which core-to-core communication is carried out with explicit messages managed by the real-time operating system (RTOS) through memory [ARINC (1997)]. When a bus is shared across a large number of cores, the bandwidth provided to each core is reduced. In such an architecture there are two types of communications: N_c-to-1 to allow cores access L2, and 1-to-N_c to allow L2 responding core requests. The most suitable NoC for this type of communications is a tree since it is specifically suited to N_c-to-1 and 1-to-N_c communications. Further, a tree NoC has been used in time-deterministic real processors [Benini *et al.* (2012),Panic *et al.* (2014)]. Our view is that tree NoCs can also be made MBPTA compliant, so that larger multicores can be used in the context of MBPTA.

In this Chapter we consider 8- and 16-core multicores. The benefits of the tree NoC are less important for smaller multicores (e.g., 4 cores) for which buses are competitive. Larger multicores (e.g., $>$ 16 cores) may fit better a clustered

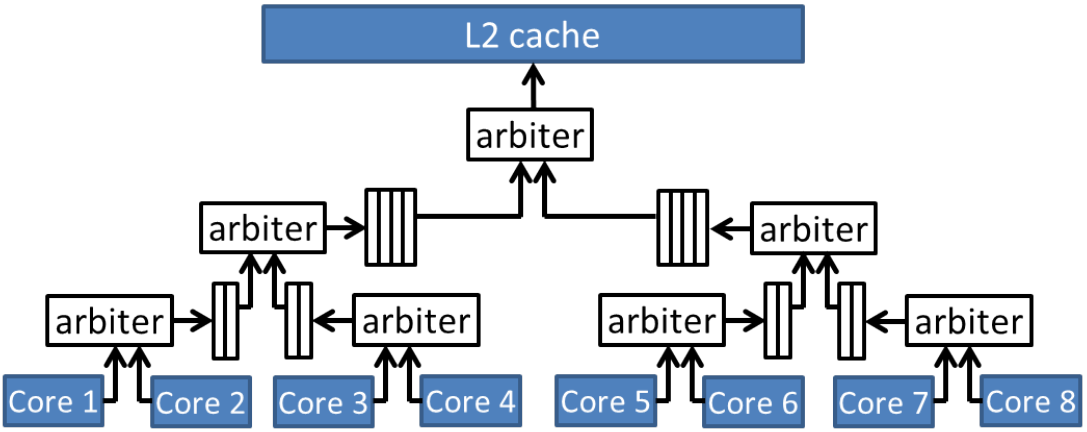


Figure 4.1: N_c-to-1 binary tree NoC for an 8-core setup

architecture where each cluster has its own local memory , since clustering provides good scalability, isolation across clusters (which is good to limit interferences) and low power, as it has been shown in [Panic *et al.* (2014)]. An alternative scalable solution would be choosing other NoC topology, such as a wormhole mesh NoC, as shown later in Chapter 5.

4.3 Single-Criticality pTNoC

Figure 4.1 shows a tree NoC to connect 8 cores to the L2. Arbiters at the bottom level, i.e. those closer to the cores, arbitrate requests from each of the two cores they are connected to. The 2^{nd} lowest level may need to buffer up to 2 requests per link (one from each of the cores below); the 3^{rd} level up to 4 requests per link, and so on. As shown, the tree allows the use of small radix routers. Each router arbitrates among the two incoming links every cycle allowing at most one request to proceed to the next level of the tree. In the last stage, the tree can deliver up to one request per cycle, which is the speed at which L2 can accept requests. Typically, the L2 requires some cycles to serve a request, but does it in a pipelined way so a new request can be accepted every cycle. Finally, requests are served through a 1-to-N_c pipelined tree where only routing is needed (no arbitration is required).

Symmetrical arbitration policies provide homogenous bandwidth assignments to the tasks running in the different cores. We analyze how the policies used for MBPTA-compliant buses [Jalle *et al.* (2014)] and already explained in Chapter 2 apply to the tree NoC. In the following explanation we assume that all links have requests ready to be arbitrated. Later in this section we describe how contention must be modeled for pWCET estimation purposes.

- With Round Robin (RR) each arbiter in the tree selects each of its incoming links alternatively in a deterministic manner. Given N_c cores, a request may be delayed by up to $N_c - 1$ requests from the other cores either because they are in front of it in a queue in the same link or because they are arbitrated first from another link in any router.

- Lottery (LOT) makes each arbiter select randomly one of the two links. As for RR, the latency for traversing the tree includes arbitration and queueing time. However, differently to RR, arbitration delay can be arbitrarily long with decreasing probabilities. The probability of waiting 0 cycles due to arbitration in one arbiter is 1/2, 1 cycle 1/4, 2 cycles 1/8, and so on and so forth. In general, the arbitration delay in an arbiter is C cycles with probability $1/2^{C+1}$.

- Random-permutation (RP) is implemented by making each arbiter produce a random permutation of two elements, 0 and 1, every two cycles, where 0 (1) indicates that the left (right) link is granted access. Thus, arbitration delay in one arbiter is 0 cycles with probability 1/2, 1 cycle 3/8 and 2 cycles 1/8 [Jalle *et al.* (2014)].

Overall, any arbitration policy valid for the bus can also be adapted for the *pTNoC* and, as we show later, *pTNoC* outperforms the bus regardless of the arbitration policy used.

4.3.1 Factoring in NoC contention

As mentioned in Chapter 1, and in other proposals in this Thesis, pWCET estimates need to be time-composable, i.e. they should hold valid regardless of the tasks running in the other cores. Time composable pWCET estimates greatly simplify incremental qualification and, by extension, help dealing with the increased timing verification and validation costs of CRTES. This is achieved by allowing each system component to be subject to formal timing validation in isolation and independently from other components.

In time-deterministic architectures measurement-based techniques rely on some hardware support, such as the *worst-case mode* [Paolieri *et al.* (2009a)]. In [Paolieri *et al.* (2009a)], each task is run in isolation at *analysis time* and hardware makes each request to experience the maximum, upper-bound, delay (UBD) that it may suffer during operation due to contention. In the case of the NoC this implies modeling the worst-case traversal time [Panic *et al.* (2014)].

We use the *worst-case mode* for time-deterministic architectures and for time-randomized architectures when deploying round-robin arbitration since contenders could issue requests systematically with specific time intervals that lead always to

the worst contention. Instead, for LOT and RP arbitration we create a *probabilistic highest contention mode (phcm)* where the task under analysis runs in isolation in one core. In all other cores dummy requests are generated so that those cores always have one request in flight in the *pTNoC*, matching the maximum load a core can put on the tree. This is the worst case because, even if contenders issue requests systematically with specific time intervals, randomized arbitration policies produce random delays on those requests and thus, change – randomly – those time intervals. Thus, the worst scenario is the one we consider, with maximum load. To that end, dummy requests are simply discarded by the L2 cache, that immediately notifies the corresponding core so that it can issue a new dummy request in the next cycle. This creates a scenario with maximum contention where requests progress randomly through the *pTNoC*. Those requests suffer contention in the L2 due to its limited bandwidth. Overall, this produces the highest (probabilistic) contention that the task under analysis can suffer during operation.

4.4 Mixed-Criticality pTNoC

Safety-related functions are assigned a safety (assurance) level that defines the steps required in the design, verification and maintenance of the hardware/software components used by those functions, which inherit functions' safety level. Safety levels are described by specific standards in each domain. For instance, in the avionics domain safety standards such as DO-178B/C[RTCA and EUROCAE (2011)] classify components into 5 different levels, from Design-Assurance Level (DAL) DAL-A to DAL-E, where DAL-A stands for the most critical level.

In the time domain, each safety-level requires providing a set of guarantees – which vary across levels – and sufficient evidence on the timing behavior of the tasks. For instance, the most critical real-time tasks (e.g., DAL-A in avionics or ASIL-D in automotive) may need strict performance guarantees provided in the form of a reliable WCET estimate.

Orthogonal to this, in each safety level, performance requirements may be heterogeneous depending on the type of application. While some critical real-time applications may need little (yet guaranteed) performance to operate on the little data read from some simple sensors, other critical real-time applications such as 3D Path Planning [Ungerer *et al.* (2013)] for Unmanned Aerial Vehicles may require high performance to process large amounts of data. Likewise, low-criticality and non real-time applications may also have heterogeneous (average) performance needs. These heterogeneous performance needs translate into heterogeneous bandwidth allocation requirements when using the NoC. In this section we describe different approaches to deal with mixed criticalities, WCET, and average performance guarantees. Our solutions provide flexible means to satisfy the

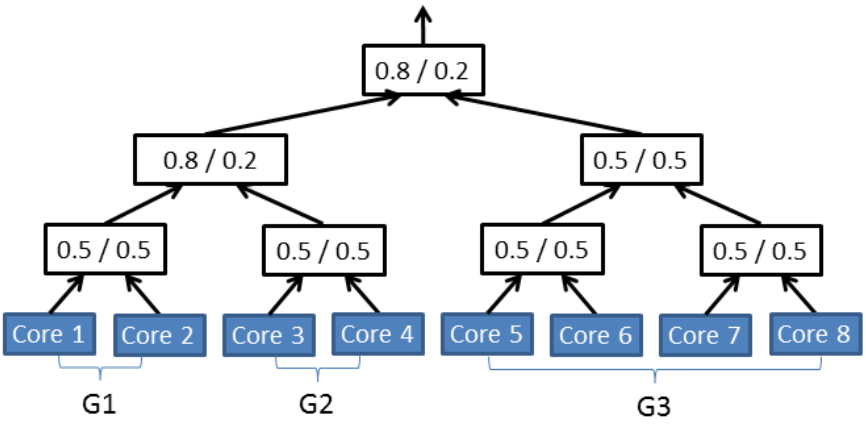


Figure 4.2: Heterogeneous guarantees in a tree NoC for an 8-core setup.

heterogeneous needs of mixed-criticality applications with a wide variety of performance requirements. For instance, let us assume we want to consolidate one DAL-A real-time task ($T1$) with commodious deadline, one DAL-B real-time task ($T2$) with a tight deadline, and six DAL-E (non-critical) tasks ($T3$ to $T8$) with no real-time constraints. In this scenario, our approach enables, for instance, allocating 20% of the *guaranteed* bandwidth to $T1$, 80% to $T2$, and remaining tasks are allowed to transmit requests opportunistically, but without timing guarantees.

4.4.1 Heterogeneous Bandwidth Assignments

Heterogeneous bandwidth allocation can be implemented by assigning different priorities to the requests of the different cores. In *pTNoC* we divide cores into several priority levels (layers). Priorities across layers may change as well as inside each layer, depending on the particular approach followed. We explore three such approaches.

(a) *Inter-layer priorities.* Each core is assigned a priority level with the requests from upper-layers prioritized over the requests of lower layers. Hence, guaranteed bandwidth can only be provided for the cores within the top layer. The bandwidth that the rest of the cores can enjoy is that left by higher-priority cores. For instance, in the avionics domain, DAL-E applications, which require only best-effort performance guarantees, are the only ones that could be assigned to a layer other than the top one.

Priorities can be implemented in each node of the tree keeping separate buffers for each priority level in those nodes where requests with different priorities may be arbitrated. The arbiter selects one request in this order: first high-priority requests from the selected link, followed by high-priority requests from the non-

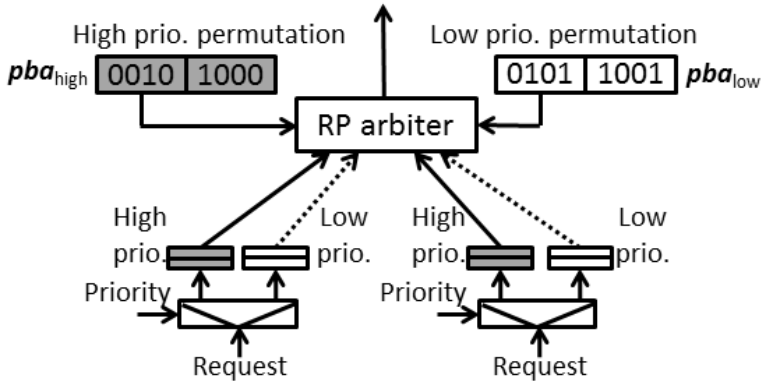


Figure 4.3: Example of an arbiter implementing inter- and intra-layer priorities.

selected link, low-priority requests from the selected link and, finally, low-priority requests from the non-selected link.

(b) *Intra-layer priorities.* Cores in each priority layer can be provided heterogeneous bandwidth allocation, which can be implemented in different ways. In the case of RP we create permutations with as many slots as needed with high-priority links being assigned more slots. For instance, one could distribute bandwidth across links as shown in Figure 4.2 in which all cores are in the top priority layer: cores 1 and 2 (Group1 or G1) get the highest bandwidth: 0.32 (0.5 in the first arbiter and 0.8 in the second and the third). Analogously, cores 3 and 4 (G2) get 0.08 of the bandwidth each and cores 5 to 8 (G3) get 0.05 of the bandwidth each. In the case of a bus, the same approach can be implemented by increasing the number of slots for cores with higher bandwidth requirements.

(c) *Mixed-layer priorities.* The two previous approaches can be combined such that all requests from cores in higher-layer priorities are prioritized over the cores in lower-layers. And within each layer bandwidth can be allocated homogeneously or heterogeneously (intra-layer priorities).

Note that intra-layer priorities, despite they can be set to allocate the bandwidth in a non-homogeneous way, provide the same level of guarantees to all cores in the layer. In that respect, while intra-layer priorities only change the bandwidth allocation, inter-layer priorities change both bandwidth guarantees and allocation.

Overall, different degrees of bandwidth (from full bandwidth to no bandwidth) can be guaranteed to the different cores, thus enabling a flexible scheme to configure the NoC to the needs of tasks in different criticality levels.

4.4.2 Implementation Remarks

pTNoC provides special purpose control probabilistic bandwidth allocation (*pba*) registers, which can be written with standard instructions in most ISAs, e.g. *mfsr* (move from special register) and *mtsr* (move to special register). How these registers are set by the RTOS is explained with an illustrative example below.

In principle, each link of each arbiter needs as many queues as priority layers. One can restrict this to having only 2 priority layers to reduce hardware overheads, especially because only the highest priority layer has true guarantees and so it makes little sense having several low priority layers.

Inter-layer priorities are implemented by prioritizing the requests in high-priority queues over the ones in low priority queues. Intra-layer priorities are implemented by allowing W-slot windows in the arbiters and using *pba* to determine which link (left (0) or right(1)) is granted access. For instance, let us assume W = 4-slot windows, which allows managing the bandwidth in 25% steps (e.g., 25% per slot) in each arbiter. In a 3-layer tree for an 8-core setup, this allows the bandwidth to range from 42.2% (75% in all arbiters, so 0.75^3) to 1.6% (0.25^3). To program the *pba* the arbiter creates 4-bit random permutations for each priority layer, see Figure 4.3. If only 2 priority layers are allowed, then 2 separate sets of *pba* registers are needed, one for each layer. In the example in Figure 4.3, for the high-priority traffic the left link is assigned 75% of the bandwidth, i.e. there are three 0's in each permutation out of four bits. For the low-priority traffic, bandwidth is divided evenly, i.e. the number of 0's and 1's is the same.

4.5 Evaluation

We model a 8/16-core processor with pipelined in-order cores. We use SoClib [Pouillon *et al.* (2009)] and gNoCsim [NanoC (2010)] simulators as explained in Chapter 3. Each core has separated first level instruction (I1) and data (D1) caches, a partitioned-across-cores L2 cache and main memory. I1 and D1 are 4KB, 8-way and 16B/line and implement random placement and replacement policies [Kosmidis *et al.* (2013a)]. The L2 is 256KB 8-way and also implements random placement and replacement policies. L2 is non-inclusive [Kosmidis *et al.* (2013b)]. D1 uses write-through miss policy, I1 is read-only and L2 is write-back. Hit/miss latency is 1 cycle for I1/D1 and 3 cycles for L2. L2 is accessed either through *pTNoC* where each arbitration stage takes 1 cycle, or through a non-pipelined bus whose latency is 3 (4) cycles in the 8-core (16-core) case. The latency values used for the tree and the bus reflect the improved scalability of the tree where arbitration is performed on a distributed manner and the maximum link length is decreased [Roca *et al.* (2012)].

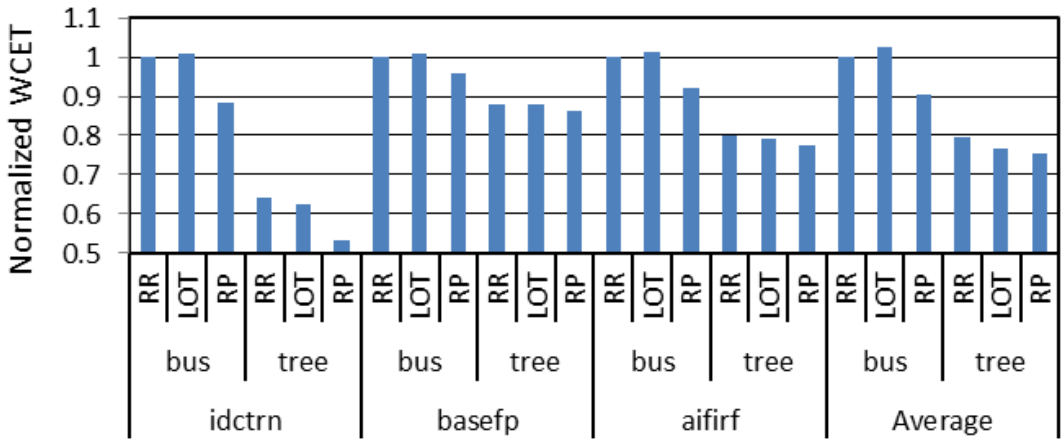


Figure 4.4: pWCET estimates for the 16-core bus and tree-based multicores normalized w.r.t. bus-RR.

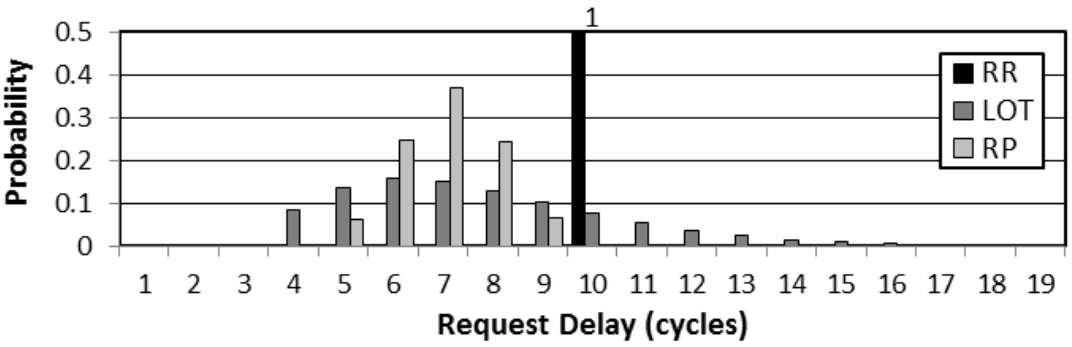


Figure 4.5: Request delay in a tree for an 8-core setup.

We make use of the proposal in [Paolieri *et al.* (2009a)] to make access to main memory be jitterless as needed for MBPTA. This further allows factoring out the variation of memory on pWCET estimates, making that NoC results can be better understood.

4.5.1 Homogeneous bandwidth setups

First we evaluate *pTNoC* under an homogeneous bandwidth setup in which all cores are provided the same level of guarantees, i.e. all cores are in the same priority layer.

pWCET estimates. We compare the pWCET estimates obtained with MBPTA for the 3 arbitration policies, namely RR, LOT and RP; for both the bus and the

tree with 16 cores. Results, shown in Figure 4.4, are normalized w.r.t. the bus implementing RR arbitration, hence, the higher the value for an arbitration policy the worse. Due to high number of different setups, we show average results across all benchmarks as well as results for few individual benchmarks corresponding to cases where differences are large (`idctrn`), small (`basefp`) and close to the average case (`aifirf`). As shown, the tree NoC always leads to tighter pWCET estimates than the bus regardless of the arbitration policy used (20% for RR, 25% for LOT, 16% for RP). This occurs because the tree has much higher guaranteed throughput due to its pipelined fashion. When comparing the pWCET across arbitration policies for the tree, we observe that RP is the best choice because its maximum and average latency to traverse the tree is equal or lower than that for the other policies.

Average performance during operation. While WCET is the most important metric in CRTES, we also evaluate average performance when real NoC traffic is experienced rather than worst-case traffic. Given that real traffic is low in the NoC, contention occurs seldom. Thus, tree NoC average execution time is only 3.4% lower than that for the bus. Differences across arbitration policies are negligible (well below 0.1% for both the bus and the tree).

Per-request tree delay. Figure 4.5 shows the per-request delay histogram to traverse the tree for the different arbitration policies at analysis time. Results correspond to the `a2time` benchmark, although all benchmarks show very similar distributions. As shown, for the RR policy its worst-case traversal time of the NoC is 10 cycles. LOT takes around 7.8 cycles on average, however, it may take any number of cycles with decreasing probability. For instance, a latency of 15 cycles is experienced around 1% of the times. Finally, RP leads to the lowest average latency (7 cycles) and it never experiences a latency higher than 10 cycles (it is 10 cycles 0.5% of the times), which is its worst case. This is the reason for RP to outperform the other policies for the tree.

4.5.2 Heterogeneous bandwidth setups

In order to study the case of mixed-criticality systems with heterogeneous performance requirements, we evaluate different setups:

1. Inter-layer priorities. First we evaluate a setup where only one core is in the high-priority layer (E1-prio) and the rest of the cores are in the low-priority layer; and also a setup in which two cores are in the high-priority layer (E2-prio), in particular cores 1 and 2, while the rest of the cores are in the low-priority layer.

2. Intra-layer priorities. All cores are in the high-priority layer with different intra-layer priorities.

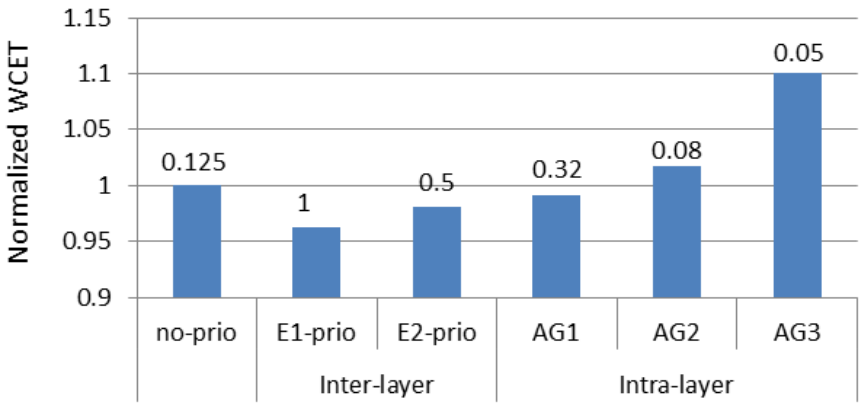


Figure 4.6: WCET estimates for the tree-based 8-core normalized w.r.t. non-priority RP case. Values on top of the columns show the guaranteed bandwidth in each case.

pWCET 8-core setup. In this case for the latter setup we consider 3 groups of cores with priorities as shown in Figure 4.2: G1 (cores 1 and 2), G2 (cores 3 and 4) and G3 (cores 5, 6, 7 and 8), with G1 having more bandwidth than G2 and G2 more than G3. We obtained pWCET estimates for each benchmark in each group (G1-3) during *analysis* time as explained in Section 4.3.1.

Figure 4.6 shows the pWCET estimates for the tree NoC for each arbitration policy normalized w.r.t. the homogeneous bandwidth (no-prio) case averaging values across all EEMBCs. The numbers on top of each column correspond to the (theoretical) guaranteed bandwidth in each case. I.e., in no-prio each of the 8 cores should get 1/8 of the bandwidth and for the E1-prio and E2-prio case each core can get full and half of the guaranteed bandwidth respectively.

In the case of E1-prio and E2-prio results correspond to the benchmark(s) running with high priority. In both cases, we observe how our proposal effectively reduces the pWCET estimate for programs running in the core(s) in the top priority. For the case of intra-layer priority we observe that cores 1 and 2 in group G1 reduce their pWCET estimate by taking bandwidth from cores in groups G2 and G3, whose pWCET estimates are affected, specially for cores in G3. In general, the differences among bandwidth allocations are relatively small. The reason is that requests quickly get to the top arbiter since buffers in each link are large enough. Therefore, contention mostly occurs in that arbiter, thus creating relatively low variability across cores sharing the same link in the top arbiter. Moreover, intra-layer priorities in this top arbiter have a much larger impact than those in other arbiters due to the same reasons: requests mostly queue in the top layer.

pWCET 16-core setup. For the 16-core intra-layer setup we used a configu-

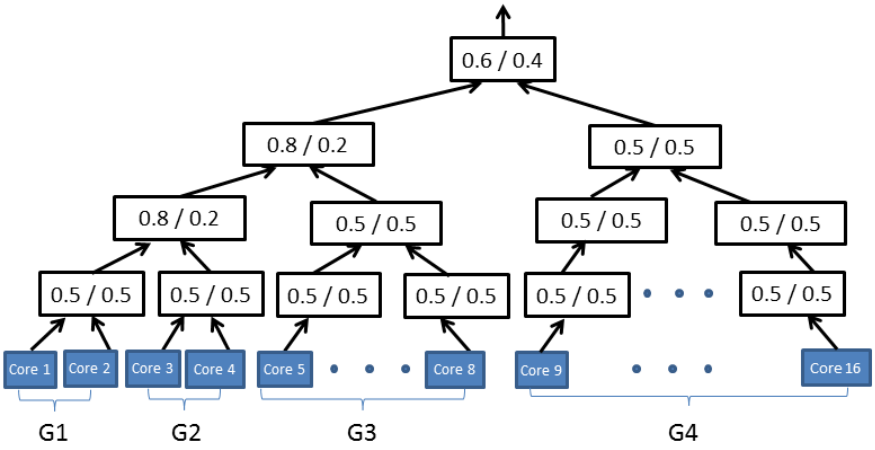


Figure 4.7: 16-core Mixed criticality setup

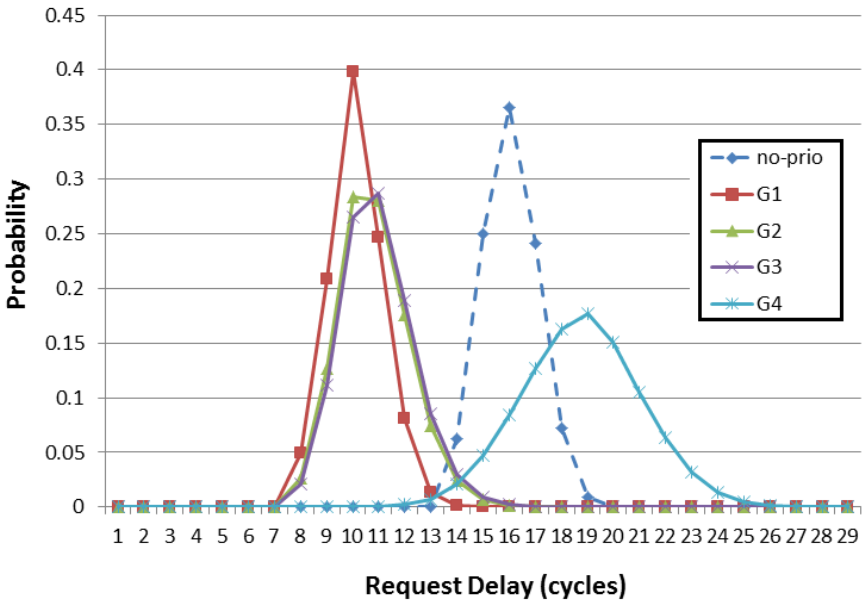


Figure 4.8: Request delay for a 16-core tree

ration similar to that for 8 cores as shown in Figure 4.7, but modifying priorities in the top arbiter to illustrate its dominant effect. The delay histogram for requests in each priority group are shown in Figure 4.8. pWCET estimates for this setup are shown in Figure 4.9. On top of each column we show the (theoretical) allocated bandwidth to each group.

Trends are analogous to those in the 8-core setup. For instance, E1-prio provides the best pWCET estimate for the core with high priority, and it is followed

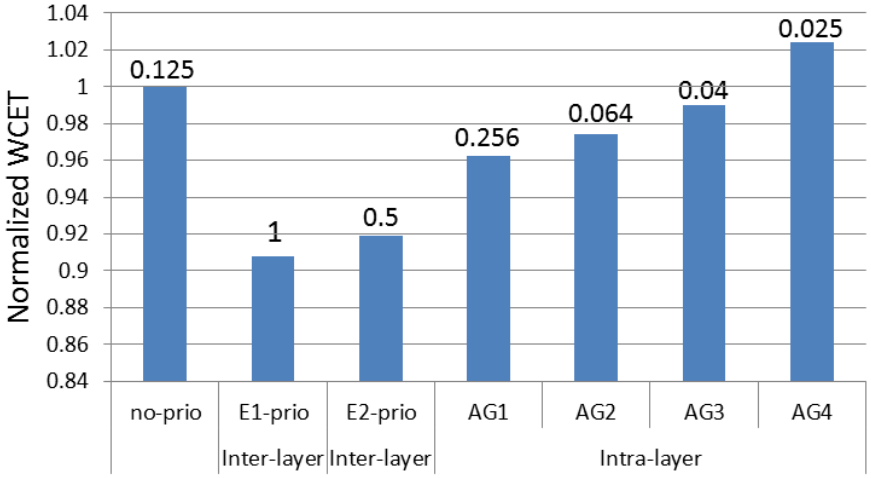


Figure 4.9: pWCET estimates for the 16-core setup for different arbitration policies normalized w.r.t. non-priority RP case.

by E2-prio for the 2 cores with high priority. Also, the groups of cores with highest G1 and lowest G4 have higher and lower pWCET estimates than the case of homogeneous priorities respectively. This correlates with the fact that the higher the bandwidth, the lower the pWCET estimate. Similar trends are observed for G2 and G3. The delay histogram in Figure 4.8 already shows that requests in G1 experience the lowest delay across groups and lower than no-prio, thus tasks in G1 obtain lower pWCET estimates than those for other cases. G2 and G3 obtain slightly worse pWCET estimates, as expected based on the delay histogram. No-prio performs a bit worse and G4 is the worst case among those, which also matches with the expectations based on the delay histogram.

It can also be observed that G2 and G3, despite having lower bandwidth than no-prio, have lower pWCET estimates. As explained before, this occurs because requests reach the top arbiter experiencing relatively low contention, but most contention occurs in the top arbiter where requests are queued until sent to L2. Therefore, bandwidth allocation in the top arbiter has much higher influence than that in the other arbiters. Since cores in G2 and G3 have higher bandwidth in the top arbiter than cores in no-prio, their pWCET is lower. Thus, pWCET differences in the 16-core setup are lower than in the 8-core case because priorities in the top arbiter are similar across links (0.6 vs 0.4) in the 16-core case, and unbalanced (0.8 vs 0.2) in the 8-core case.

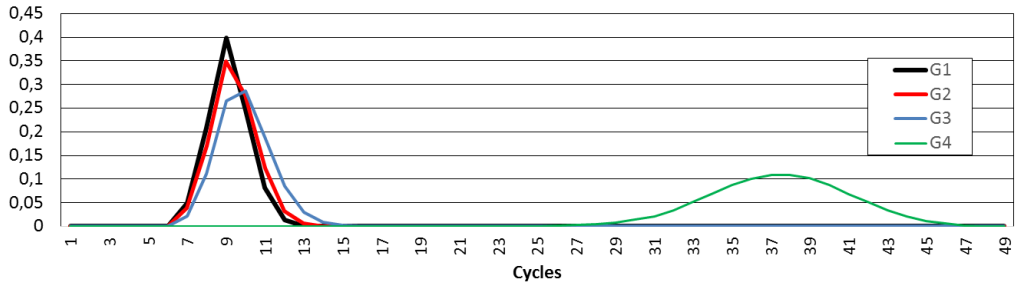
We could have used a different bandwidth distribution in the top arbiter in the 16-core case (e.g., 0.8 vs 0.2). This would have increased the bandwidth for cores in G1, G2 and G3 at the expense of reducing it for cores in G4. However, even if the theoretical bandwidth for cores in G1 is much higher than that in G3 (25.6%

for G1 cores vs 4% for G3 cores) real bandwidth is very similar for cores in G1 and G3 and performance differences are completely negligible (around 1% in terms of pWCET estimates). As explained before, by allowing requests to proceed to the next arbiter as long as they were granted access, made them reach the last arbiter very quickly. The main reason is that buffers were virtually-unlimited. By varying the bandwidth in the top arbiter this limitation could not be removed.

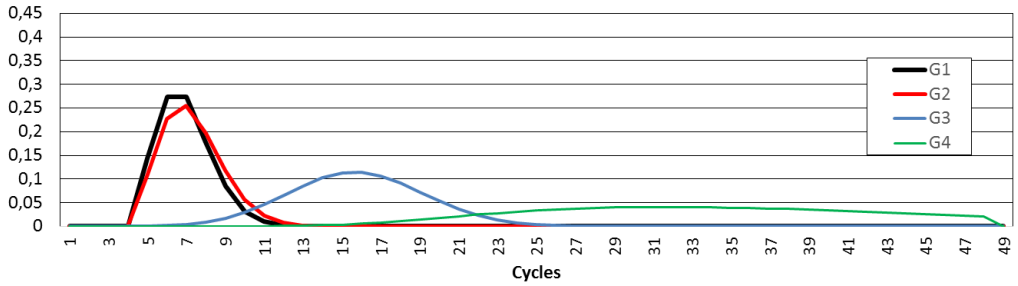
Thus, another direction to investigate a fine-grained control of the bandwidth allocation in the *pTNoC* is the impact of the size of the buffers. For that purpose we have obtained the histogram of delays for requests in each group (from G1 to G4) for two different buffer sizes (virtually-unlimited, so size 8, and minimal, so size 2). Note that a buffer size of just 1 entry would not allow back-to-back requests arriving through the same tree port to be processed immediately, thus introducing 1-cycle bubbles in between. Thus, the minimum buffer size considered is 2 entries. The histogram for the virtually-unlimited buffer size is shown in Figure 4.10a. As expected, the delay histogram for cores in G1, G2 and G3 are almost identical, where G1 is slightly better than the others and G3 slightly worse. When reducing buffer sizes to only 2 entries, heterogeneous arbitration in lower-level arbiters becomes more relevant and this reflects in the histogram of delays. This is shown in Figure 4.10b. We observe that differences between G1 and G2 are still negligible. By having 2-entry buffers, requests from G1 and G2 (up to 4) reach quickly the two top-level arbiters where up to 4 requests can be stored from their part of the tree. Hence, contention in the bottom-level arbiters is still very low and so, heterogeneous arbitration there is still ineffective. But there is bigger difference for requests from G3, and G4 requests obtain latencies with a much wider spread.

The impact of the different buffer sizes on pWCET is shown in Figure 4.11. Results have been normalised w.r.t. the case where bandwidth is allocated homogeneously across cores and the buffer size is 8. We observe that pWCET estimates for G1 and G2 improve slightly when decreasing buffer size due to the improved efficiency of the bandwidth allocation scheme with lower buffering capabilities. This latter fact also reflects in G3, whose pWCET estimates grow since now its effective bandwidth is lower due to having to compete against G1 and G2 requests in the third-level arbiter, where G3 has lower bandwidth allocated. Finally, cores in G4 get slightly higher pWCET estimates despite having lower average latency for requests. This stems from the fact that execution time variability is higher in some cases due to the larger latency range for requests. Still differences w.r.t. large buffers are low.

Average performance. We have also evaluated average performance when running under no-prio, inter- and intra-layer priority configurations. Figure 4.12 shows the average performance of all tasks in the workload for the 16-core setup.



(a) Virtually-unlimited buffer size (8)



(b) Minimal buffer size (2)

Figure 4.10: Histogram of delay for requests for different heterogeneous groups

pTNoC effectively provides both, heterogeneous bandwidth guarantees and allocation, with very reduced impact on average performance. For the inter-layer configurations the slowdown is less than 5.5%. Differences come from the case where several programs with high memory bandwidth requirements fall in the same workload, some of them being in the top priority cores (thus creating plenty of high-priority traffic) and others starving systematically in the low priority cores. For the intra-layer configuration giving higher bandwidth to some cores negligibly improves average performance across all cores w.r.t. the no-prio setup. Average performance is just 2% better for G1 w.r.t. no-prio, 0.5% worse for G4 w.r.t. no-prio and 0.4% better across all groups.

4.5.3 Implementation and Energy Results

We have implemented the proposed tree design using the 45nm technology open source Nangate library [Inc. (2014)] with Synopsys DC. We have used M1-M4 metallization layers to perform the Place&Route with Cadence Encounter of the different arbitration nodes. To determine the placement of the different arbitration nodes we have followed the approach proposed in [Roca *et al.* (2012)] to minimize link length. Table 4.1 summarizes the implementation results for 8- and 16-core

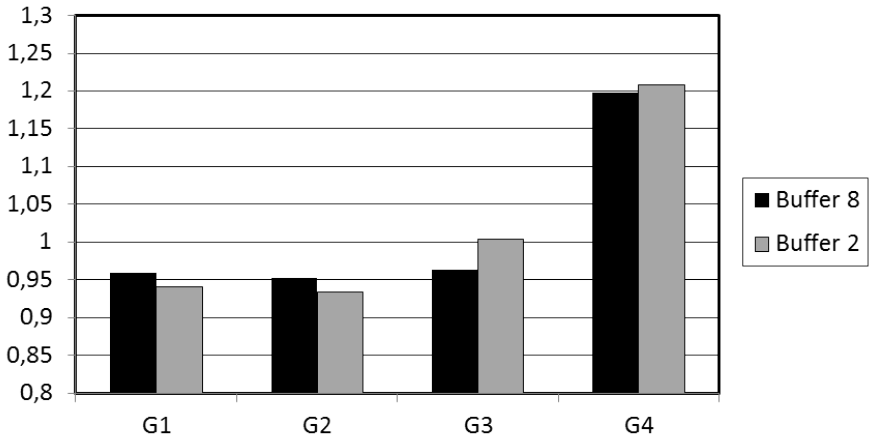


Figure 4.11: pWCET estimates of EEMBC normalised w.r.t. homogeneous bandwidth and unlimited buffer size in the arbiters.

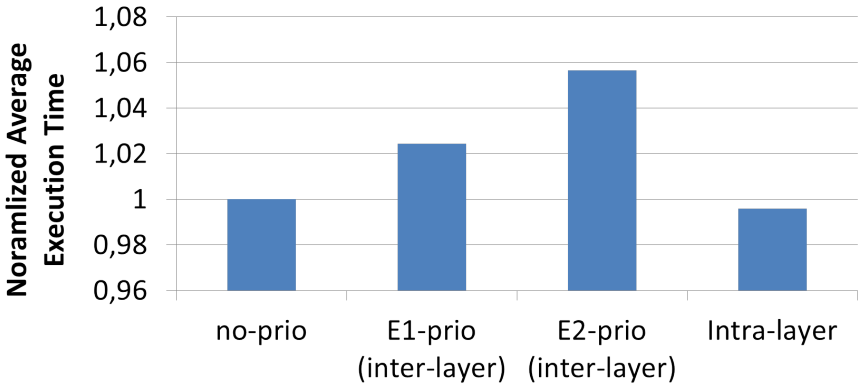


Figure 4.12: Average execution times for 16 cores for different arbitration policies.

trees using two priority layers. Results for a bus implementation are also shown for comparison purposes. Area and delay results in the table are normalized w.r.t. an 8-core mesh NoC. As shown, tree NoCs require lower area than the 2D mesh network because the tree NoC implements only the resources required to perform all-to-one communication. However, as expected, the area required for implementing the tree is larger than that required for the bus. Critical path delay results shown in the table determine maximum achievable frequency of the different NoC designs. Maximum achievable frequency mainly depends on two factors: network size and arbitration inputs. For the bus and the tree, network size impacts operation frequency as it determines the maximum link length required in the floorplan. The number of arbitration inputs or contenders is given by the number of layers used to perform the arbitration. As shown in Table 4.1 *pTNoC* shows good scal-

Table 4.1: Results of the synthesis normalized w.r.t. 8-core mesh values. Absolute values are also provided for completeness.

	Area					
	8-core			**16-core**		
	mesh	tree	bus	mesh	tree	bus
Relative	1	0.54	0.38	2.30	1.55	1.24
Absolute ($10^4 \mu m^2$)	18.4	10.0	7.0	42.3	28.6	22.9

	Delay					
	8-core			**16-core**		
	mesh	tree	bus	mesh	tree	bus
Relative	1	0.90	1.33	1	0.96	1.61
Absolute (ns)	0.83	0.75	1.1	0.83	0.80	1.34

ability up to 16 nodes unlike the bus that shows a delay 1.67× and 1.47× worse than the *pTNoC* for 8 and 16 cores, respectively. Moreover, while the number of arbitration layers penalizes maximum achievable frequency, its impact can be contained if only two arbitration layers are used as proposed in this Chapter. Note that results shown in the table are for the case of RP. However, results for RR and LOT arbitration, also computed but not shown in the table, are roughly the same.

We have also computed energy values when executing several applications with the proposed NoC design. In particular, we have computed values for EEMBC benchmarks and for one synthetic benchmark. The synthetic benchmark is a corner case where high number of misses in L1 occur to create high contention in the NoC. Differences in energy measurements for regular benchmarks (EEMBC) are imperceptible while RP behaves slightly better (1.5%) in the case of the synthetic benchmark. The conclusions we extract from these measurements is that the very low implementation overhead of RP has negligible impact in energy values even in the least favorable scenario (low contention) while the reduction in the execution time provided by RP provides slightly lower energy values in the most favorable scenario (high contention).

4.6 Comparison of tree-based manycore architectures

In this Chapter we have already seen that it is possible to obtain tight WCET estimates with the time-randomised tree-based NoCs and how those designs are suitable for mixed-criticality CRTES. In this section we integrate these designs in a MBPTA compliant multicore and perform an in-depth comparison of time-

deterministic and time-randomized tree-based setups in terms of WCET estimates.

Tightness of WCET estimates on time-deterministic and time-randomised platforms has only been compared in single-core setups [Abella *et al.* (2014)]. Those platforms have been chosen for the sake of allowing a fair comparison, despite targeting suboptimal platforms. In this section we identify hardware setups meeting the requirements of each approach while being comparable. This basically implies using the same hardware setup varying placement and replacement policies in caches, as well as arbitration policies in shared resources such as the NoC and the memory controller. Next we explain hardware setups for both paradigms and what is necessary for obtaining WCET estimates.

4.6.1 Time-Deterministic platform

Timing Analysis Technique

In the case of the Time-Deterministic architecture, we use Measurement Based Deterministic Timing Analysis (MBDTA) to obtain WCET estimates. In this regard, time-composable bounds to traverse shared resources have to be factored in when deriving WCET estimates. As shown in [Paolieri *et al.* (2009a)], the architecture needs to guarantee that the maximum delay a request will suffer has an Upper-Bound-Delay (UBD). UBD is a function of the number of contenders and access time to the shared resource. Once the UBD has been calculated, we need to enforce the shared resource to have a fixed latency of exactly UBD cycles on each access during the analysis phase.

Tree NoC

In the case of the tree NoC, we use round-robin arbitration. Given that it is fully pipelined, the UBD corresponds to the *zero-load latency (zll)* – so the latency to traverse the NoC with no contention – plus 1 cycle per contender, which is the maximum delay that each other core could produce on each request of the Task Under Analysis (TuA) in one of the arbiters. In our fully-pipelined tree, the zll matches the number of levels of the tree, which is $\lceil \log N_c \rceil$. Therefore, the UBD of the tree NoC is as follows:

$$UBD_{noc} = (N_c - 1) + \lceil \log N_c \rceil \tag{4.1}$$

Memory controller

Likewise, for WCET estimation purposes, the memory controller latency is assumed to match the number of contenders multiplied by their access latency, which

is assumed to be fixed. For instance, if the core count is N_c and the memory controller latency to accept a new request is L_{mc}, then the UBD – the highest delay that could be experienced due to contention – of the memory controller before processing a new request of the TuA is as follows:

$$UBD_{mc} = (N_c - 1) \cdot L_{mc} \tag{4.2}$$

Cache memory

Both L1 and L2 cache memories implement time-deterministic placement and replacement policies, i.e. modulo placement and LRU replacement. During the analysis phase, in the time-deterministic paradigm, for cache memory we have to either (1) provide worst case alignment of the memory objects, which is an open problem or (2) fully control what will happen at analysis and operation time. The first solution is expensive and not usually not doable[1]. Thus, we assume the second solution, in which we guarantee that the TuA will have the same memory placement at deployment and at analysis. Note that this assumption is very optimistic and plays in favor of the time-deterministic paradigm against which we compare our approach.

In both cases (deterministic and probabilistic setup) we use hardware way-based cache partitioning technique (columnization) [Chiou *et al.* (2000)] to prevent inter-task interference in L2 cache space.

4.6.2 Time-Randomized platform

Timing Analysis Technique

In order to derive WCET estimates, in the case of the Time-Randomized platform we use MBPTA as described before in Chapter 3. During the analysis phase we probabilistically upper bound the behaviour of all components.

Tree NoC

We use the tree-based NoC with random-permutation symmetrical arbitration policy, as we already shown its superior performance compared to LOT and RR arbitration. Moreover, as explained before, we use *probabilistic highest contention mode (phcm)* during the analysis phase, as needed for obtaining WCET estimates.

[1] In fact controlling memory placement, and so cache placement, is one of the main difficulties to obtain reliable WCET estimates with MBDTA.

Table 4.2: Processor setup

DL1, IL1 configuration	16KB 4-way 32B/line
DL1, IL1 latencies	1 cycle hit/3 cycles miss
DL1 policies	write-through, write-no-allocate
IL1 policies	read-only
L2 configuration (per-core)	32KB 4-way 32B/line
L2 latencies	2 cycles hit/7 cycles miss
L2 policies	write-back, write-allocate, non-inclusive
DTLB, ITLB configuration	16 entries fully-associative
Tree NoC latency	1-cycle per level
Memory latency	16 cycles
Memory inter-request delay (L_{mc})	27 cycles

Table 4.3: Processor configurations evaluated

Core count	**4** cores	**8 cores**	**16 cores**			
Time-randomised	RAN4	RAN8	RAN16	RANcache16	RANnoc16	RANmc16
Time-deterministic	DET4	DET8	DET16			

Memory controller

The time-Randomized memory controller we use in this comparison implements a random-permutation policy [Jalle *et al.* (2014)], with separate queues for each core. During the analysis phase we need to assume that every core will have the request ready in every arbitration round, so to guarantee that we upper bound the behaviour at operation time.

Cache memory

For the L1 and L2 cache memory we use random placement [Hernández *et al.* (2016) and random-replacement function [Kosmidis *et al.* (2013a)]. As mentioned before, L2 cache memory is partitioned across cores.

4.6.3 Evaluation

Experimental Setup

We consider different setups. Time-randomised and time-deterministic ones are referred to as RAN and DET respectively. We consider 4-core, 8-core and 16-core setups. In the case of 16 cores, we also evaluate some setups where all components are time-randomised except one: either caches, the tree NoC or the memory

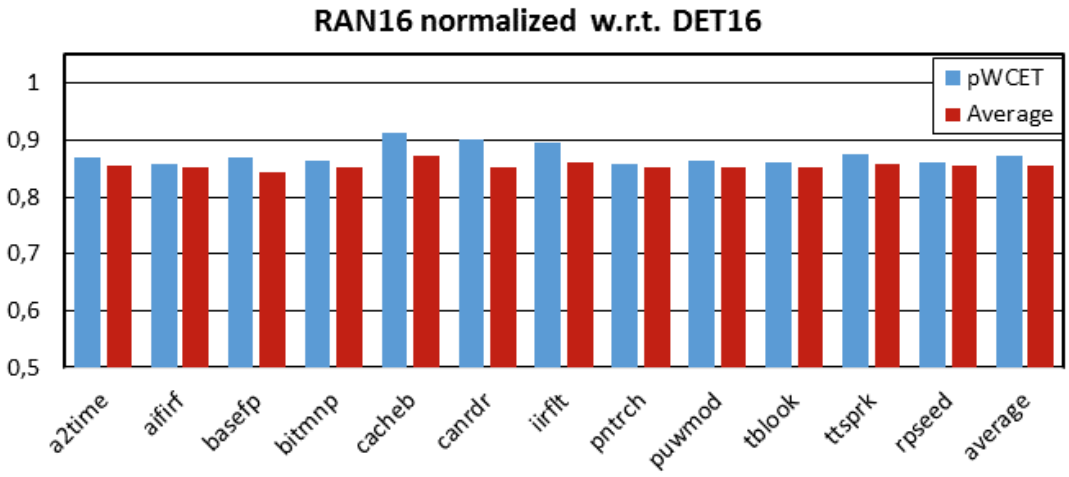


Figure 4.13: pWCET for RAN16 normalised w.r.t. DET16.

controller are time-deterministic. For instance, whenever all components are randomised except caches, we refer to this setup as RANDcache16. The complete list of setups evaluated (and pairs compared) is shown in Table 4.3.

Evaluation Results

Next we compare the different time-deterministic and time-randomised setups. We perform the analysis in two axes: (1) from the core count perspective and (2) from the resource perspective. First we compare the two approaches with a varying number of cores to observe how they scale. Then, we fix the 16-core setup as the baseline and compare the time-deterministic setup against different versions of the time-randomised one where one of the resources is made time-deterministic to discount its effect and understand how each resource impacts WCET.

Core count scalability analysis

Results for the time-randomised setups with 16, 8 and 4 cores are shown in Figures 4.13, 4.14 and 4.15 respectively. In the case of the 16-core setup we provide both, pWCET estimates and average execution time. As shown, pWCET estimates are 13% lower for RAN16 than for DET16. If we further account for the potential impact of other memory placements, then benefits can only grow since RAN16 randomises cache placement, thus making memory placement irrelevant. Instead, WCET estimates for DET16 are only valid as long as the memory placement analysed is preserved during integration (and during operation). Otherwise, the impact of other potential placements should be accounted for, further increas-

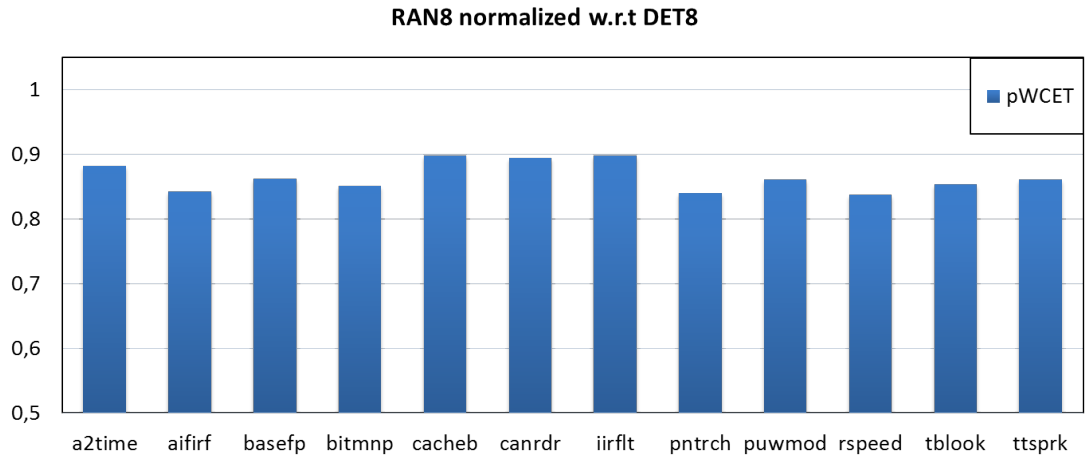


Figure 4.14: pWCET for RAN8 normalised w.r.t. DET8.

ing WCET estimates. Also, it can be noted that average execution time and pWCET estimates for RAN16 are typically within a narrow 2-3% range.

When analysing other core counts (4 and 8), we observe that benefits of RAN4 and RAN8 slightly diminish w.r.t. those of RAN16, since pWCET estimates are 1-2% closer to those of their time-deterministic counterparts. One would expect benefits being lower with lower core counts. However, with 4 cores L2 bandwidth is highly utilised due to frequent store instructions and the use of a write-through DL1. Therefore, by adding extra cores, increased utilisation creates delays for all cores regardless of the arbitration policy. Hence, the relative gains cannot increase noticeably.

Resource sensitivity analysis

In Figure 4.16 we show results for different setups, as explained in section 4.6.3. All results are normalised to the baseline DET16 setup, in which all components are time-deterministic. RAN16 is included for reference. We observe that making cache deterministic makes little difference since random modulo performs almost identically to modulo placement. Therefore, variations between RAN16 and RANcache16 relate mostly to the statistical variability across samples that has some influence on pWCET estimates. Still, RAN16 shows some (minor) benefits w.r.t. RANcache16 since random variability introduced by the different components is offset more easily when further sources of variability are introduced.

Instead, RANnoc16 shows that by making the NoC fully deterministic, benefits w.r.t. DET16 disappear. This occurs because L2 accesses are very frequent and hence, using upperbounded latencies instead of time-randomised ones increases

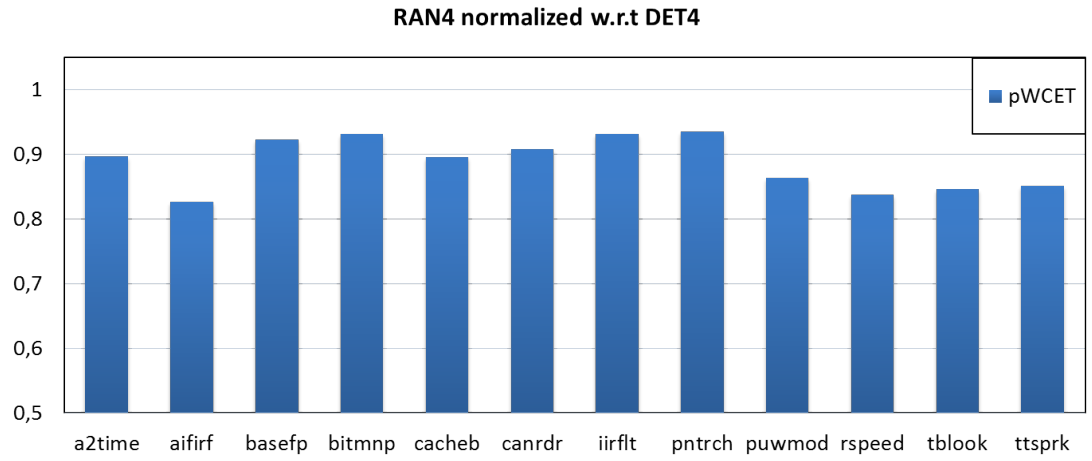


Figure 4.15: pWCET for RAN4 normalised w.r.t. DET4.

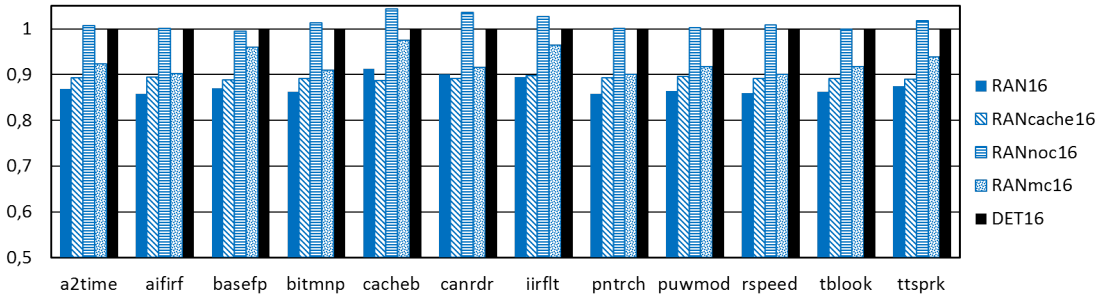


Figure 4.16: pWCET for different 16core setups normalised to RAN16 setup.

the latency for all those frequent requests. The fact that pWCET estimates for RANnoc16 are slightly higher than WCET values for DET16 relates to the fact that MBPTA provides necessarily pWCET estimates higher than actual measurements. Since measurements in both setups are roughly identical on average, then RANnoc16 is slightly worse than DET16 in terms of WCET estimates.

Finally, RANmc16 shows that making memory arbitration policy deterministic has also some impact in WCET estimates. Still, such impact is not huge in general since most programs hit in L2 and thus, memory is accessed seldom. However, some programs access memory more often (e.g., `cacheb`), and thus their pWCET estimates increase due to the higher memory latencies.

All in all, we conclude that the NoC is the most sensitive component in this setup due to its high utilisation.

4.7 Conclusions

MBPTA has emerged recently as a powerful method to derive WCET estimates for critical tasks in safety-related systems. Multicore designs providing the properties needed by MBPTA have been presented in the literature, but they rely on a shared bus to communicate cores and memory, and do not fit well mixed criticalities.

We have presented a MBPTA-compliant tree NoC, *pTNoC*, that outperforms buses in the 8- and 16-core setups evaluated. *pTNoC* enables the realization of mixed criticalities on multicores by managing different guarantee levels and bandwidth assignments with minimum impact on average performance.

Our results for 16 cores show 16% to 25% average WCET reductions for our tree w.r.t. the bus for different arbitration policies. Further WCET reductions of up to 9% are obtained when using priorities for mixed criticalities. Also, our results show that *pTNoC* incurs low area and energy costs.

Time-randomised architectures for manycores lead to reduced pWCET estimates w.r.t. those obtained for time-deterministic setups. We have shown that gains are highly independent of the core count as long as one shared resource – the NoC in our case – is highly utilised. The impact of this high utilisation is further illustrated in the per-resource analysis where we show that time-randomised caches perform very close to time-deterministic ones, whereas using randomised access policies for shared resources provides some significant gains in terms of WCET.

Chapter 5

PTA-compliant mesh NoC

Wormhole-based NoCs (wNoCs) are widely accepted in high-performance domains as the most appropriate solution to interconnect a high number of cores[1] on chip. However, wNoCs suitability and ability to deliver low WCET estimates in the context of critical real-time applications has not been demonstrated yet. In this Chapter, in the context of probabilistic timing analysis (PTA), we propose a PTA-compatible wNoC design that provides tight time-composable contention bounds. The proposed wNoC design builds on PTA ability to reason in probabilistic terms about hardware events impacting execution time (e.g. wNoC contention), discarding those sequences of events occurring with a negligible low probability. This allows our wNoC design to deliver improved guaranteed performance.

5.1 Introduction

Wormhole-based NoCs (wNoCs) are deployed in high-performance domains to connect a high number of cores on-chip. However, wNoCs efficient use in the context of CRTES applications has not been shown yet. Unlike buses or other existing centralized network architectures, wNoCs perform the arbitration of communication flows in a distributed manner, which severely complicates the derivation of request contention bounds as required in real-time domains. In this line, some works show that, while reliable contention upper bounds can be provided for Commercial off-the-shelf (COTS) wNoCs [Rahmati *et al.* (2013),Panic *et al.* (2016b),Panic *et al.* (2016a)], those bounds are pessimistic, preventing an efficient use of high-performance wNoCs for CRTES.

wNoC bounds are pessimistic because, whenever timing events can lead to the stall of a request, they are assumed to occur systematically, and hence factored in

[1]Since CRTES industry has only certified single-core and few multicore systems, we assume a high number of cores ≥ 16

the derived contention bounds. At the NoC level, since many different flows with different criticality levels might potentially be contending for different resources, e.g. router ports, timing analysis techniques are forced to make the pessimistic assumption that all contenders will simultaneously request the same resources. A simple and intuitive way to reduce contention bound pessimism consists in getting information about when and where communication flows in the wNoC will occur such that the exact interference that requests experience can be reliably and tightly factored in. Unfortunately, obtaining this low-level information is not only out of the ability (and will) of end users, but it also breaks time composability. The lack of time composability occurs because one task's load on the wNoC affects the worst-case execution time (WCET) estimates of its corunners, with devastating consequences in (incremental) system integration: any change in a task requires reanalyzing all other tasks (i.e. performing regression tests), which ultimately results in prohibitively high integration costs. Even worse, the WCET of a critical task could depend on the accuracy of the information obtained for a lower criticality task.

In this Chapter we propose several wNoC designs based on a new randomized wormhole router design, which makes that the contention in the network has a probabilistic behavior compatible with MBPTA requirements, thus leading to reduced contention bounds. The contributions of this Chapter can be summarized as follows:

1. We integrate efficiently random permutation arbitration [Jalle *et al.* (2014)] in wNoCs routers to avoid systematic bad behavior and make them amenable for MBPTA.

2. We show that, while limiting the number of in flight requests in deterministic wNoCs[1] does not help reducing contention bounds, it helps reducing significantly those bounds in a probabilistic wNoC and thus, improves WCET estimates.

3. We propose an alternative mechanism to control the injection of packets in the wNoC based on controlling the frequency at which requests can be injected in the network that outperforms regular injection limitation schemes in high contention scenarios while requiring similar hardware complexity for the network interfaces.

[1] In this Chapter we refer to a deterministic NoC as the one with a round-robin arbitration.

Table 5.1: Summary of the main symbols used.

Symbol	Description
Z_{ll}	zero load latency
$\overline{WCD}$	Time-composable upper-bound to contention delay
F_i	Packet stream traversing the same source-destination route and requiring the same grade of service along the path.
H_i	Number of hops in a flow F_i
R_i^j	Router (hop) j in a flow F_i (see Figure 5.2(a))
r_i^k	Packet (request) k in a flow F_i
NR_i^j	Number of queues that can potentially contend for an output port that F_i is targeting at R_i^j
$\omega(i,j)$	Function that returns the index x of the worst possible destination flow F_x that starts at the hop R_i^{j+1} and reaches the worst possible destination in terms of indirect blocking of packets of flow F_i

5.2 Problem Formulation

Deterministic time-composable bounds in COTS wNoCs are pessimistic since we cannot make any assumption on the contenting flows and, therefore, we need to assume that all flows will produce the highest interference. In this section we describe the target setup and we show why contention bounds are pessimistic.

5.2.1 Network Baseline

We consider a mesh network topology as it is the most common topology used in wNoCs, though the analysis presented in this section and the wNoC designs proposed in this Chapter are also suitable for other network topologies (e.g. torus). The symbols used in this Chapter are summarized in Table 5.1.

In our reference mesh wNoC configuration, each node comprises a *PME* (Processor/Memory element) and a router that communicates with the other nodes. The PME can be either a processor core, main memory, I/O, etc. In the network, several traffic flows (F_i) may be active at the same time. Each node can be identified using (x,y) coordinates and the router located at coordinates (x,y) is referred to as $R(x,y)$.

In a wNoC, the routing algorithm determines the path that a packet follows within the network, and consequently, the number of routers or *hops* (H), a given flow requires to move from a source to a destination node. Hence, a router can also be identified as R_i^j, in which j represents the hop j of flow F_i, when moving towards its destination. Therefore, routing determines the flows that potentially contend with F_i at every router in its path. Deterministic routing has been shown to provide

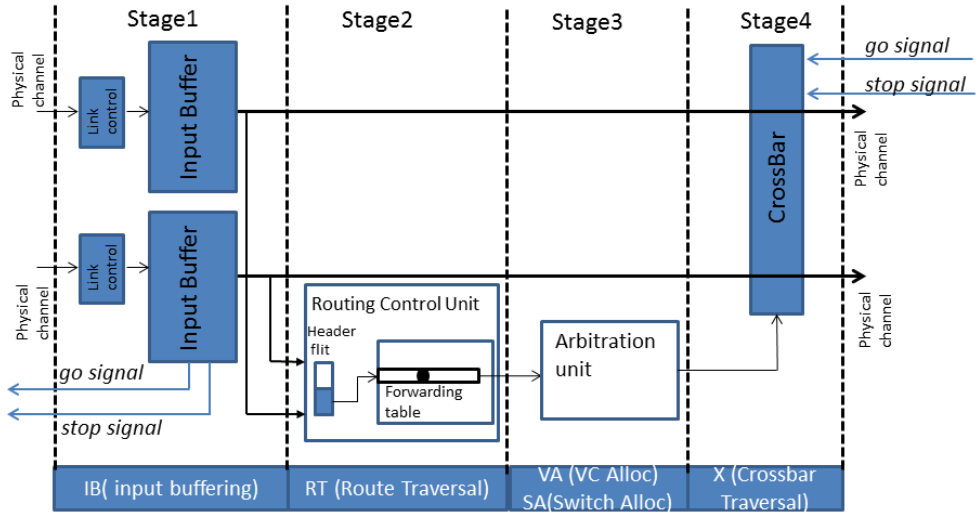


Figure 5.1: Router stages.

time analyzability [Rahmati *et al.* (2013)]. We use XY routing, as it is the preferred deterministic routing algorithm for regular NoCs due to its low implementation cost, although a similar analysis is possible for any other deterministic routing policy. With XY routing, packets are forced to use the X dimension first: In the X dimension the position of the target node w.r.t. the source node determines whether to go right (X+) or left (X-) direction. The same approach is used for the Y dimension. Once a packet is routed using the Y dimension, it cannot be forwarded back to the X dimension. These routing restrictions determine the maximum number of flows contending with F_i at a given router for an output port (NR_i^j).

Communication flows comprise multiple NoC packets. A packet is the minimum arbitration unit in the network and it can be split into one or several *flits* (short of control flow units). The first flit of a packet is called header flit and contains the information required to forward the packet to the destination. We refer to the k-th packet generated by flow F_i as r_i^k.

A typical wormhole router comprises several modules including input buffers (IB), routing (RT), virtual-channel allocation (VA), switch allocation (SA), and crossbar traversal (X) modules. Figure 5.1 shows the canonical architecture of a wormhole-based mesh router and the different stages of the router pipeline. When a flit of a packet has enough space in the input buffer, the *go signal* of the link-based flow control allows the incoming flit to be stored in the router input buffer.

Routing modules determine the router output port based on the destination

bits included in the control information of the header flit. Once the destination is known, the target port is requested in the arbitration module. Then, based on a given arbitration policy, the router arbiter decides which packet is granted access to the output port. The majority of COTS wNoCs use round-robin to arbitrate amongst packets requesting access to a given output port. Arbitration only occurs at the packet level and for the header flit. Once a connection is established between an input and output port, it remains until the tail of the packet leaves the router. At this moment a new arbitration can be performed in case other requests are also requesting this output port.

5.2.2 Contention in the wNoC

The latency experienced by a packet to traverse the network in the absence of contention is referred to as *zero-load latency* (*zll*). However, contention may cause the header flit to get stalled. When this happens, the remaining flits of the packet get also stalled and latency experienced by a given packet is higher than *zll*.

The first thing to consider when computing the contention in the network is the number of flows that will be actually contending for the different shared resources. In our case, as we are after time-composable contention delay bounds, no assumptions can be made on the particular active flows in the wNoC. That is, it is assumed that any node in the network is entitled to send and receive packets from any other node. Similarly, when computing the contention delay for a packet, we assume that, by the time it is injected in the network, any other potential contending flow is also active at that moment, transmitting its packets in a way that it produces the worst possible contention scenario. In order to reproduce the worst possible contention scenario we need to consider the *worst direct contention* and the *worst indirect contention* [Kim *et al.* (1998)].

Let us illustrate the process of measuring contention in the wNoC with an example. Let us consider a 3x3 wNoC setup like the one shown in Figure 5.2. We want to measure the worst contention experienced by a packet P_i of flow F_i. F_i is the flow originated at the node attached to $R(0,0)$ with destination the node attached to $R(0,2)$. At R_i^1, first router, a request r_x coming from port $X-$ might be potentially contending with P_i for the same output port and in the worst-case r_x will be granted access first. However, due to the backpressure flow-control it is not guaranteed that at the time r_x is granted access it will leave the router as the input buffer of the next router ($R(0,1)$) might be occupied. In the plot $R(0,1)$ input buffer is occupied by r'_x. In general, to measure contention it is required to iterate from the destination node backwards to analyze the time that is required by a packet to leave a given router. That is, to have a slot available in an $R(0,1)$ input buffer, we need to consider the time that r'_x requires to leave $R(0,1)$ that in turns depends on the time r''_x needs to leave the input buffer at $R(0,2)$. Equation

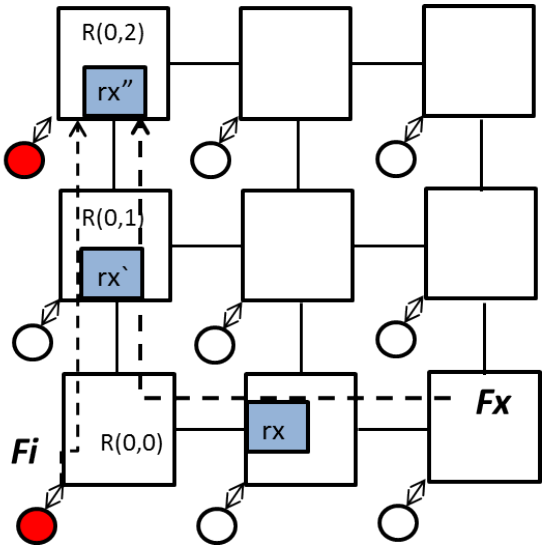


Figure 5.2: 3x3 Mesh.

5.1 was proposed in [Panic *et al.* (2016b)] and provides a general formulation for the worst contention experienced in wNoCs under the scope of time-composability.

$$\overline{WCD_i} = \sum_{j=1}^{H_i} \left((NR_i^j - 1) \times \prod_{m=1}^{H_{\omega(i,j)}} NR_{\omega(i,j)}^m \right) \tag{5.1}$$

In the equation above, $H_{\omega(i,j)}$ is the number of hops required by the worst possible destination flow ($F_{\omega(i,j)}$), i.e. the flow that creates the worst contention to the flow under analysis[1]. The first multiplicand $(NR_i^j - 1)$ corresponds to the *contention introduced by the round robin arbitration* in each of the routers that the flow F_i traverses. The second multiplicand $\prod_{m=1}^{H_{\omega(i,j)}} NR_{\omega(i,j)}^m$ corresponds to the *indirect contention delay* in each hop due to the worst possible destination flow $F_{\omega(i,j)}$.

The first question that raises from the contention formulation above is whether the assumptions on top of which this model is built are pessimistic or not. We want to know if considering that nodes in the network are injecting packets in an uncontrolled manner or, in other words, that the number of in-flight requests per node in the network is unlimited, is the reason why composable contention delay bounds are pessimistic [Panic *et al.* (2016b)].

Interestingly, as already shown in [Panic *et al.* (2016b)], limiting the injection

[1]The worst possible destination depends on the routing algorithm as well as on the actual number of ports that routers have. Sometimes it matches the farthest destination but this is not necessarily always the case.

of the flow under analysis, has no impact on $\overline{WCD}$ since this only affects intra-task contention and not the contention due to inter-task interferences. However, from a global perspective considering that nodes in the network are only allowed to have a limited number of requests in flight may limit the contention. For example, if only one packet per core is allowed, a packet P_i will only be contending with P_x once and thus, potential conflicts will be reduced.

Let us reuse the previous example to analyze the impact that reducing the number of in-flight requests has in the computed bounds. To do so, we consider the most favorable scenario where only one in flight request per core is allowed. In this 3x3 network setup (Figure 5.2) F_i experiences contention at 3 different points that correspond to the 3 input buffers that F_i traverses to reach the destination node. At $R(0,0)$ two possible requests might be contending with P_i, the ones originated at sources attached to $R(1,0)$ and $R(2,0)$. At $R(0,1)$ packets potentially contending with P_i are the ones originated at $R(1,1)$ and $R(2,1)$. As only one packet per flow is allowed if a request is contending with P_i at $R(0,0)$ then the very same request cannot be at the $R(0,1)$ input buffer. Still, at $R(0,1)$ requests from two different nodes are allowed. Moreover, the worst possible situation is that those packets contending with F_i get ejected from the network as soon as they are not contending with F_i packets so a new packet can be injected in the network and contend again with F_i.

With the previous example we have illustrated that despite limiting the number of in-flight requests reduces the potential conflicts this does not necessarily allow reducing contention bounds in the general case. The reason is that time-composable WCET estimates require considering the worst possible interleaving of requests in the wNoC and this causes, in general, worst situations to be also possible when allowing only one request in flight per-flow. In Section 5.4 we corroborate this hypothesis with contention measurements.

5.3 Probabilistic wNoC Designs

Unlike deterministic wNoCs, probabilistic network designs do not require the timing analysis to consider that all accesses systematically experience their worst possible contention. Therefore, the probabilistic analysis made by MBPTA arises as a suitable approach to reduce the pessimism factored in the contention in wNoCs. To enable the derivation of pWCET estimates with MBPTA, two conditions must hold in the wNoC design: (i) conflicts in the wNoC must have a probabilistic nature (i.e. should occur with a given probability); and (ii) the execution conditions (contention) under which the timing measurements of the application are collected at *analysis* are actually an upper bound of those that will occur during *operation*. Condition (i) requires modifications in the arbitration unit of the router (Sec-

tion 5.3.1) and condition (ii) requires defining a contention scenario which safely upper-bounds the worst possible one (Section 2.1.2). In this section we present how a COTS wNoC must be adapted to enable the derivation of tight pWCET estimates with MBPTA.

5.3.1 MBPTA-compliant wNoC Router Design

To make a wNoC design MBPTA-compliant, we have to make packet jitter to follow a probabilistic behavior. To do so, hardware changes are required in the arbitration unit of the NoC router. An intuitive, but inefficient, MBPTA-compliant policy to grant access to a given output port is to simply select one of the requests randomly. This might cause a given request to take long (in theory infinite) time to be granted access. Instead, from the different MBPTA-friendly arbitration policies, we choose random permutations as it delivers superior performance and bounded contention [Jalle *et al.* (2014)]. Random permutations grant access to N contenders in a round-robin fashion, but in a random order. Such order changes every N arbitrations, so that each contender is granted access once every N slots, but in a random order.

To implement random permutations in the wNoC router, we modify the arbiter to be able to generate a random permutation P_i of all four inputs for every output port, where the four inputs and the output port belong to the group $(X+, X-, Y+, Y-, In)$. Whenever one or more packets request access to a given output port, the arbiter grants access according to P_i and an arbitration pointer. When a permutation is generated the arbitration pointer points to the first input port in the permutation. If the first input in the permutation is not requesting the output port then the next input port in the permutation is selected. This process is repeated until an input port with a pending request is selected. At this point the arbitration pointer is moved to the input port in the permutation after the one granted access. When the pointer reaches the end of the permutation, a new random permutation window is generated.

When a new permutation is required, we generate a probabilistic arbitration window (paw) comprising two arbitration windows (permutations) with all inputs (paw_{low} and paw_{high}). In this way, the generation of a new permutation occurs when the pointer reaches the end of paw_{low} so that the new permutation is available for the next arbitration. At that point, paw_{high} becomes paw_{low} and the new permutation becomes paw_{high}. This avoids generating an arbitration window at the same time that one of its elements needs to be selected. Figure 5.3 shows how the proposed random arbiter works.

Hardware implementation. Random permutations can efficiently be implemented as shown in [Jalle *et al.* (2014)]. We implement them for each of the 5 output ports using $N = 4$ inputs, so that they need only $N \cdot log_2 N$ bits for the reg-

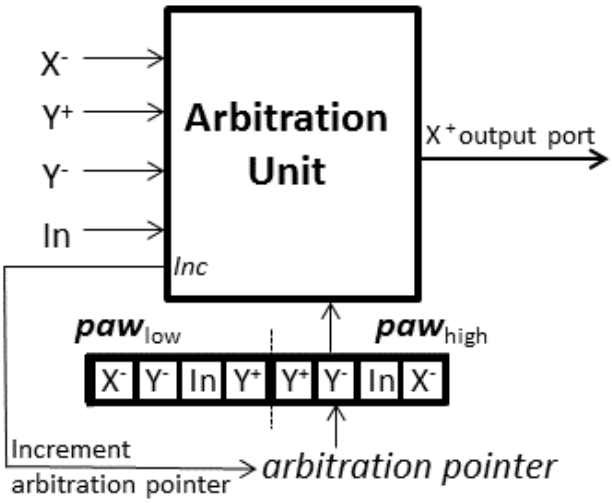


Figure 5.3: Arbiter.

ister and $N-1$ random bits. Those random bits are produced by a pseudo-random number generator as the one proposed in [Agirre *et al.* (2015)].

This router is the basic component on top of which the two probabilistic wNoC designs proposed later in this Chapter rely on. The random arbitration performed in this router allows introducing delays in the measurements to probabilistically represent the contention in the wNoC. While arbitration decisions are random, they carry out dependences across arbiters since the actual requests contending for a given output port in a router often depend on the (random) decisions taken in other routers. Any state of the wNoC in terms of contention moves to any other state with a given probability due to the purely random nature of all arbitration choices. Therefore, each sequence of states occurring during the execution of the task under analysis occurs with a given probability and hence, each potential execution time has a true probability to occur, as needed to apply MBPTA.

5.3.2 Reducing Contention in Probabilistic wNoCs

By combining worst-contention scenarios with the probabilistic router architecture proposed in section 5.3.1 we can produce execution conditions during analysis that upper-bound those during operation. Execution time measurements collected under this setup can be used reliably to apply MBPTA in order to derive WCET estimates. However, if we do not anyhow limit the contention in the network, the stalls experienced by the requests of the task under analysis can be very high and thus, WCET estimates will account for high contention for all requests, similarly to the case of time-deterministic wNoCs. In time-deterministic wNoCs the

worst-case contention with, for instance, round-robin arbitration, is accounted for all requests regardless of the degree of contention in the network. In the case of probabilistic wNoCs, requests experience the actual contention of the worst-case scenario modeled at analysis, which is enforced not to be exceeded during operation. Therefore, decreasing maximum contention by design opens the door to obtaining lower WCET estimates with probabilistic wNoCs, as already shown for tree wNoCs in Chapter 4.

In order to decrease contention and derive tighter WCET estimates in the wNoC, we propose two mechanisms: (1) Limiting the number of in-flight requests or (2) limiting the injection frequency. The first approach is more suitable for applications that are sensitive to network latency while the latter for applications with high throughput requirements. It is important to mention that reducing the contention by reducing the number of requests in the network is suitable for probabilistic wNoCs because, in such designs, requests interleave probabilistically and therefore, worst-case alignment of flows and arbitration decisions do not need to be accounted for systematically (as opposed to the case of time-deterministic wNoCs). In other words, already proposed techniques for reducing contention in time-deterministic wNoCs, such as injection throttling [Thottethodi *et al.* (2001)], cannot obtain tighter WCET estimates.

Limiting the number of in-flight requests (LNR)

Contention in the network can be reduced by limiting the number of requests in-flight for all the nodes in the network. With our proposed router design, we remove the need to know the exact alignment and we only need to ensure that, during the analysis phase, the task under analysis can have *up to* n requests in-flight and all the other cores have *always* exactly n requests in flight. In this case, execution times obtained for the task under analysis are obtained under worst possible contention conditions.

Limiting injection frequency (LFR)

In this case, to be able to derive tight contention delay bounds, we propose to control injection frequency at wNoC nodes. Similarly to the previous proposal, during the *analysis* phase the task is run in isolation. Requests of the task under analysis are allowed to be injected if at least Minimum Inter-request Delay (MID) cycles have elapsed since the previous request was injected. For all the other cores, we have to generate requests at the maximum allowed frequency, i.e. once every MID cycles to create maximum contention.

Since we do not assume any specific pattern and maximum contention is enforced during analysis, time-composability is preserved. During operation all cores

can inject requests with the same restrictions imposed during the analysis (at least *MID* cycles after the previous injection).

Hardware support

LNR can be easily implemented at the network interfaces by having a counter for the number of requests in the network. This counter is increased when a new request is injected in the wNoC and it is decreased once a response from an injected request is received back at the node the task under analysis is attached to. If such hardware mechanism is in place, the generation of the maximum contention scenario at analysis can be implemented in software by using microbenchmarks that constantly send requests to the node that affects the most the flows of the core under analysis.

For the hardware implementation of LFR we need at the network interface a counter from 0 to up to MID-1 cycles to control the maximum frequency at which requests (from contenders and the core under analysis) are injected in the network. Stressing scenarios can be reproduced by software means as in the previous mechanism.

5.4 Evaluation

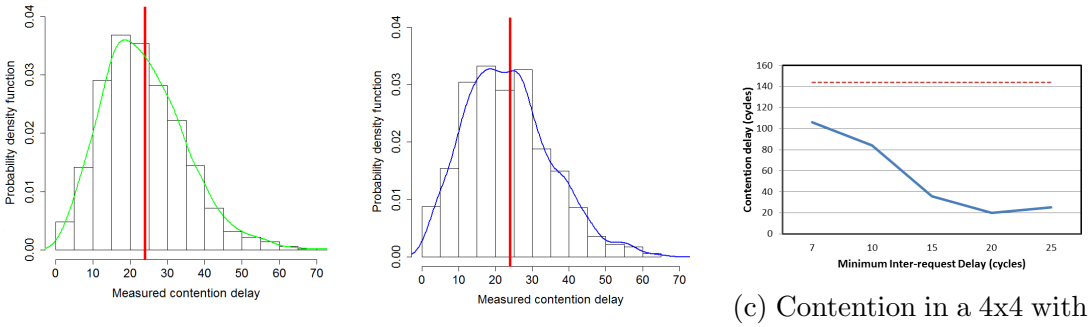
5.4.1 Methodology

Target Processor Architecture

We model a wNoC-based manycore processor with pipelined in-order cores with a simulator based on the SoCLib simulation framework [Pouillon *et al.* (2009)] as explained in Chapter 3. Each core has separated first level instruction (I1) and write-through data (D1) caches, a partitioned-across-cores write-back L2 cache and main memory. I1 and D1 are 16KB, 4-way and 16B/line and the L2 has 128KB 4-way per core to discount L2 cache effects from the analysis. All caches implement random placement and replacement policies [Kosmidis *et al.* (2013a)]. L2 is non-inclusive [Kosmidis *et al.* (2013b)]. D1 uses write-through miss policy, I1 is read-only and L2 is write-back. The processor includes a 2-entry store buffer allowing the pipeline to progress while having available entries. Hit/miss latencies are 1 and 3 cycles for I1/D1 and 2 and 7 cycles for L2.

wNoC

We model the wNoC with an enhanced version of the gNoCsim [NanoC (2010)] simulator, as explained in Chapter 3. Cores and memories are connected using



(a) 1 request 3x3 (LNR) (b) 5 requests 3x3 (LNR) (c) Contention in a 4x4 with LFR

Figure 5.4: Contention in 3x3 and 4x4 wNoC setups with LNR and LFR.

a mesh network topology with XY routing. For a NxN mesh we index routers from R(0,0) to R(N-1,N-1). The shared L2 cache memory and a shared memory controller are connected at router R(N-1,N-1). The memory controller implements random permutation policy. Two virtual networks are used to split requests and responses. Routers are pipelined and consist of 4 stages: input buffer, routing, switch allocation, and crossbar traversal. In line with other works [Panic *et al.* (2016b),Panic *et al.* (2016a)] in all wNoC setups we use single-flit packets only to improve performance guarantees. The number of VCs is 1. Additional virtual channels would not provide higher guaranteed performance in our setup, as discussed in [Panic *et al.* (2016b)].

We compare our probabilistic wNoC designs with [Panic *et al.* (2016b)], since it is the only wNoC setup that allows the derivation of composable WCET estimates. The rest of the platform features are kept identical in both setups, the deterministic one and the probabilistic, to understand the differences in the guaranteed performance provided by the wNoC since both setups are compatible with MBPTA.

Workload

For characterizing wNoC performance we use synthetic traffic. Worst-contention scenarios are created by artificially injecting requests in the wNoC targeting worst-possible destinations and at the maximum allowable rate. As representative real-time workloads we use the EEMBC Autobench suite [Poovey *et al.* (2009)].

5.4.2 Characterizing wNoCs Performance

We have used synthetic traffic to characterize the performance of the probabilistic wNoC. Figure 5.4 shows the measured contention when nodes have 1 (a) and

Table 5.2: Minimum guaranteed throughput w.r.t ideal case. DET refers to a deterministic setup.

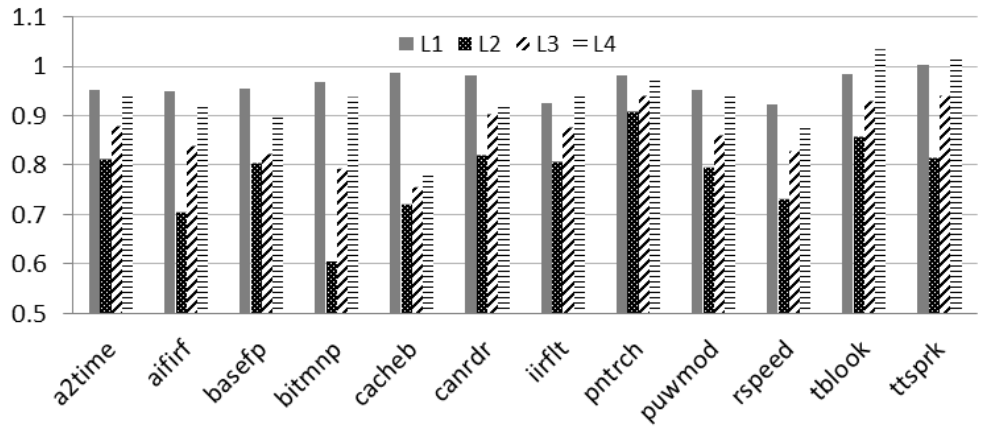
	3x3	4x4	6x6
LNR	0.333	0.119	0.118
LFR	0.856	0.795	0.327
DET	0.333	0.111	0.007

up to 5 (b) requests in a 3x3 network setup for the flow that goes from $R(0,0)$ to $R(2,2)$. As shown, contention in a probabilistic wNoC setup follows a probabilistic distribution. This probability distribution is centered around $\overline{WCD}$ (the worst contention in a deterministic wNoC setup). Interestingly, the shape and location of the distribution is almost identical regardless of the number of in-flight requests, one or five.

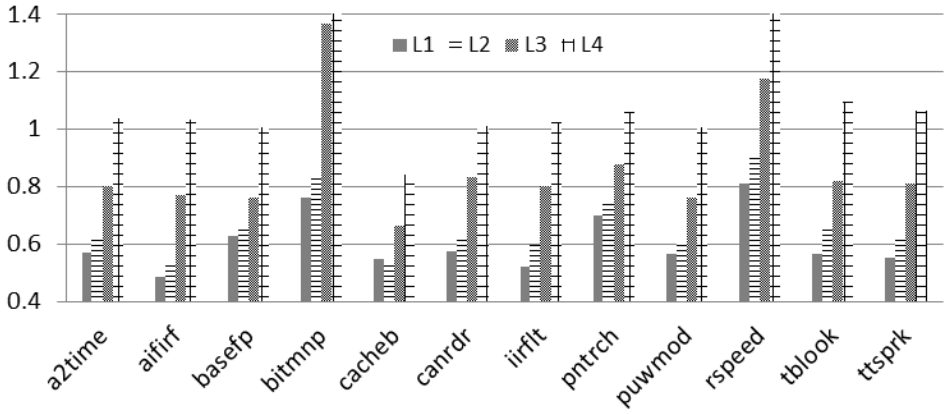
The conclusions we draw from this analysis are twofold. First, as shown in [Panic *et al.* (2016b)], worst-contention situations in the wNoC are also possible when allowing only one request in-flight per core. Second, probabilistic wNoCs do not allow reducing network contention per se as in average the worst contention experienced in the network remains the same. Note that having contention delay following a distribution is caused by the way arbitration windows in the different routers align. Sometimes the interleave with random arbitration windows makes requests progress fast and sometimes makes them get stalled longer.

Figure 5.4c shows the results of an additional experiment that analyzes to what extent the contention in the network can be reduced by controlling the frequency at which network interfaces issue requests. As shown, average contention delay for a 4x4 mesh decreases for our probabilistic wNoC as we decrease injection frequency flattening when the total injection rate of the mesh leads to a non-saturated network so that traffic effects decrease. Contention starts to decrease at an injection frequency of 1 request every 16 cycles per core, so that all 16 cores inject at most 1 request per cycle in total on average, which is the maximum ejection rate of the network when all nodes target the same destination. On the other hand, with decreasing injection frequency we also postpone requests of our core, so then we need to find a good balance between low contention and large delay for our requests.

Table 5.2 shows minimum per node guaranteed throughput normalised w.r.t. the ideal scenario in which a node in a NxN wNoC setup is able to eject flits from each of the nodes at a 1/(NxN) rate. As shown in the table, both probabilistic setups outperform guaranteed throughput values provided by the deterministic wNoC setup and LFR is the one able to retrieve more bandwidth. The deterministic wNoC setup provides very poor bandwidth guarantees for big and medium



(a) 3x3



(b) 4x4

Figure 5.5: LNR pWCET estimates normalised w.r.t. a deterministic wNoC (Ln means up to n requests in-flight allowed).

size wNoCs because of the distributed nature of the arbitration, which allocates a very small fraction of the bandwidth to the farthest nodes when the network is fully congested [Panic *et al.* (2016a)].

5.4.3 Performance Evaluation

For evaluating the performance guarantees of our probabilistic wNoC setups, we use the EEMBC single-threaded workloads as the task under analysis. In these experiments, the task under analysis is placed at the node attached to $R(0,0)$ and the rest of the cores are forced to cause worst-possible contention as described in Section 5.3. Figure 5.5 shows the pWCET estimates achieved by limiting the

number of in-flight requests (**LNR**) for the different benchmarks. Results are normalized w.r.t. the case of the deterministic wNoC. All the existing task communications target the shared memory controller that is attached to $R(2,2)$ in the 3x3 case and to $R(3,3)$ and $R(5,5)$ in the 4x4 and 6x6 case[1]. Other possible task placements provide similar comparative results. As shown in Figures 5.5 (and 5.7a), **LNR** wNoC setup outperforms the performance guarantees achieved by the deterministic wNoC design.

It might not seem obvious the reason why **LNR** behaves better than a deterministic setup since, as already shown in Figure 5.4, requests in both setups experience the same average contention. The reason lies in the way contention is distributed. For a deterministic approach, where a particular alignment of requests cannot be assumed, it needs to be considered systematically that requests will be absorbed at a constant rate that is equal to $\overline{WCD}$. On the contrary, in a probabilistic approach roughly 50% of the requests will get absorbed faster than $\overline{WCD}$. These fast requests make possible to take advantage of the store buffer of the pipeline more frequently than for a deterministic approach and allow a higher overlapping between computation and communication, thus leading to smaller execution time[2]. In particular, the behavior of the deterministic wNoC is a *fill-and-stall* behavior of the store buffer in front of store bursts, thus stopping pipeline progress always. Conversely, the probabilistic wNoC allows releasing store buffer entries sometimes earlier due to lower contention delay, and for the time a new store arrives at the store buffer, there is space available so that the pipeline keeps progressing in parallel with the processing of stores in the wNoC.

Figure 5.6 shows pWCET estimates for the **LFR** approach for 4x4 network setups. Note that the plot shows only results for three benchmarks representing the best, the average and the worst case. As we can see, the improvement achieved by the **LFR** approach is very significant being 40% on average for the case of 20 Minimum Inter-request Delay cycles. If we move to a 6x6 mesh, **LFR** brings huge guaranteed performance benefits: in Figure 5.7b we see that we have 93.3% improvement on average for $LFR50$. This occurs because latency bounds grow linearly with distance with **LFR**, whereas they grow exponentially for time-deterministic approaches, that need to account for the worst potential behavior for all requests, which is extremely unlikely and far above the typical case for **LFR**. If we compare **LNR** and **LFR** for the 6x6 setup, we observe that **LNR1** provides pWCET estimates 75% lower than those on a deterministic network, but this is still almost 4x higher than **LFR**, which is clearly the best choice as the size of the mesh grows.

[1] Although our approaches scale smoothly regardless the core count, we do not consider larger manycores due to the increasingly poor scaling of deterministic wNoCs for larger core counts.

[2] This holds for timing-anomaly-free processors like the one used in our experiments.

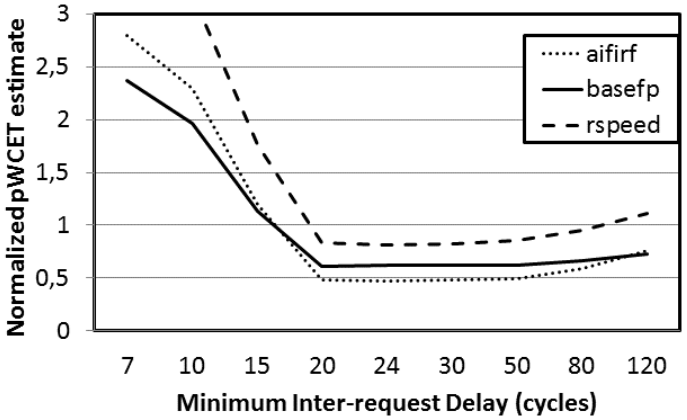


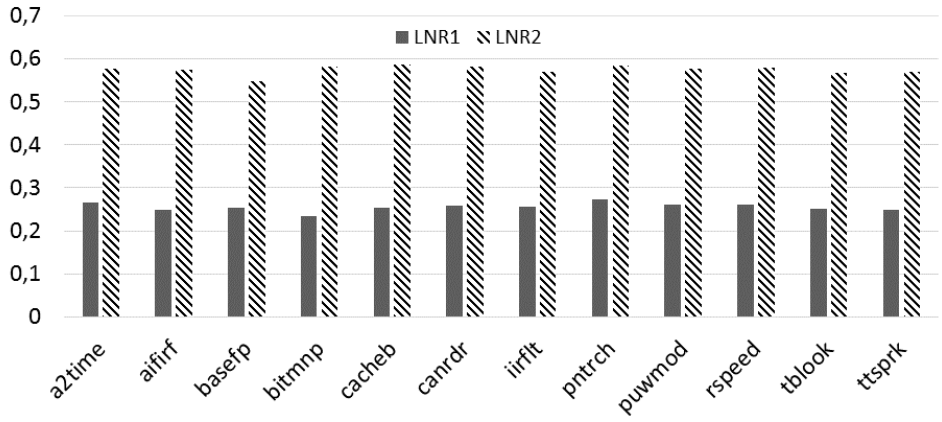
Figure 5.6: LFR pWCET estimates w.r.t a deterministic wNoC.

A direct comparison of results for both approaches for 3x3 and 4x4 network setups is shown in Figure 5.8. In this plot we present results for the the best possible configuration in each case. Note that the best configuration, the number of in-flight requests in the case of **LNR** and the actual frequency of requests for the case of **LFR** providing the best performance, only depends on the properties of the task under analysis and not on the actual load that co-runners put in the network.

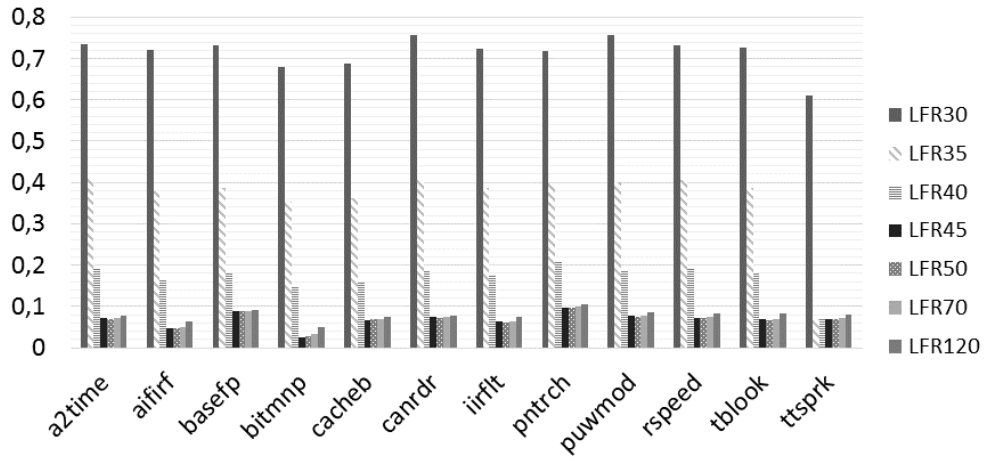
Finally, we have measured how far **LNR** and **LFR** pWCET estimates are from the ideal case. To do so, we compare the results of probabilistic wNoC setups with the results for tasks running in isolation, so experiencing always zero-load delay. On one hand, for small networks **LNR** achieves pWCET results that are relatively close to the case without any contention, on average 9% for the 3x3 setup and 34% for 4x4, whereas for 6x6 it is around 330%. On the other hand, when increasing the size of the network the performance cost of the **LFR** mechanism drops significantly: 35% and 32% for 3x3 and 4x4 setups respectively, and only 5% for the 6x6 setup w.r.t. no contention since requests are much less likely to contend anywhere in the NoC when the NoC size increases. Therefore, **LNR** and **LFR** are complementary solutions fitting small and large NoCs respectively.

5.5 Related work

We classify the existing research in NoCs for real-time applications in the following four categories: (1) Real-time specific NoCs, (2) NoC calculus, (3) analytical worst-case bounds, and (4) probabilistically analysable interconnects. Our work fits in the 4^{th} category and represents the first general realization of wNoC designs



(a) Limiting Number of Requests



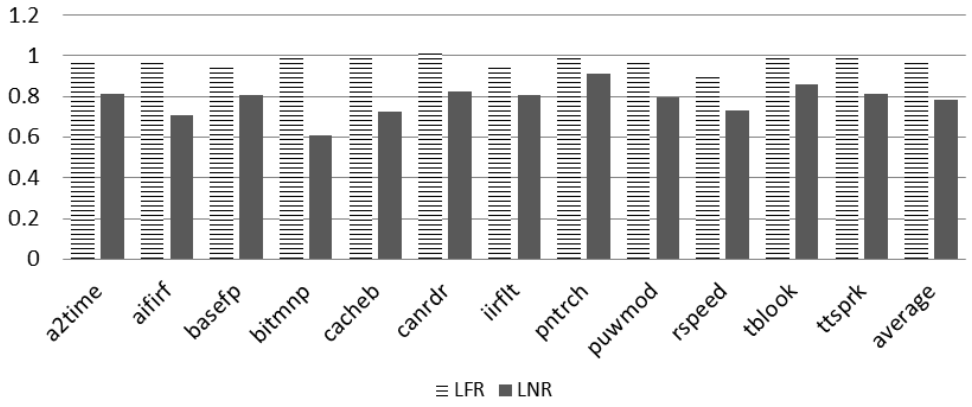
(b) Limiting injection Frequency of Requests

Figure 5.7: **LNR** and **LFR** for 6x6 mesh normalized w.r.t. a deterministic wNoC.

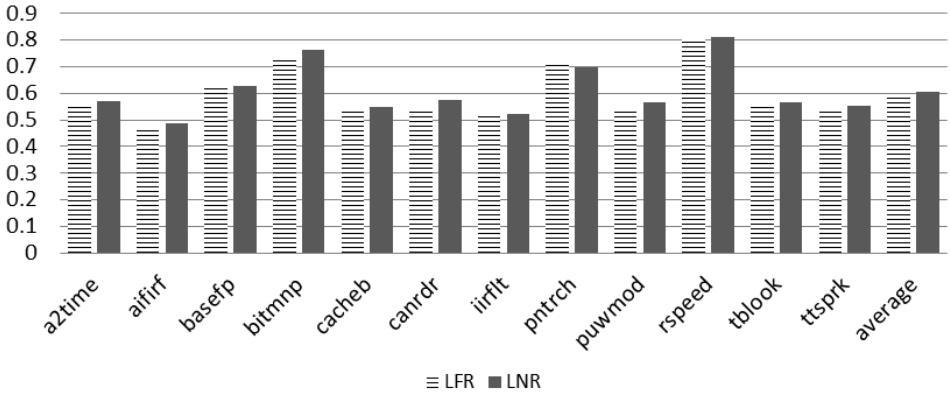
for probabilistic time analysis. Main differences across the four approaches are summarized in Table 5.3.

Real-time specific NoCs. We distinguish two main groups in this category. (1a) NoCs using TDMA [Goossens *et al.* (2005)] allow an easy derivation of composable WCET estimates but require specific designs only suitable for real-time applications that provide poor average performance. (1b) Approaches using flit-level virtual-channel (VC) prioritization [Shi & Burns (2008)] to bound the contention in the NoC by forwarding first flows with higher priority. The latter approaches require significant router modifications, abundant VC resources and break time composability because details of all flows are needed.

Network Calculus. Works based on Network Calculus [Le Boudec & Thiran



(a) 3x3



(b) 4x4

Figure 5.8: **LNR** and **LFR** performance comparison

(2001)] abstract communication flows using arrival curves that upper-bound the amount of traffic within any time interval. With the upper-bounded arrival curve and lower-bounded service curve, delay bounds can be derived. Time composability is lost with this approach unless worst possible traffic conditions are considered and (2a) deterministic – rather than (2b) stochastic [Lu *et al.* (2014)] – delay bounds are used [Yue Qian & Dou (2009)]. However, these assumptions defeat the whole purpose of Network Calculus. Thus, Network Calculus is appropriate when traffic information available during analysis is accurate, which may be for off-chip traffic, but is generally unaffordable for on-chip traffic.

Analytical wNoCs bounds. Another set of works focuses on determining wNoC packets worst-case traversal time (WCTT) by (3) considering worst-case conditions, first assuming limitations on the packet-injection rate [Lee (2003)], and

Table 5.3: Approaches for achieving NoC performance guarantees. Category (4) includes the wNoC proposed.

	Bounds Computation	Composable WCET	HW Changes	Performance	Real-time
(1a)	Analytical	Yes	Deep	Low	Hard
(1b)	Analytical	No	Deep	Moderate	Hard
(2a)	NoC Calculus	No	None	High	Hard
(2b)	NoC Calculus	No	None	Very High	Soft
(3)	Analytical	Yes	None	Low	Hard
(4)	Not needed	Yes	Moderate	High	Hard

later without this limitation [Rahmati *et al.* (2013),Ferrandiz *et al.* (2012)]. Finally, authors in [Panic *et al.* (2016b)] showed that measuring inter-task interferences in a wNoC using the Worst Contention Delay ($\overline{WCD}$) metric results in much tighter WCET estimates than using WCTT.

MBPTA compliant interconnects. (4) MBPTA compliance has been achieved for bus designs [Jalle *et al.* (2014)] either using Lottery arbitration [Lahiri *et al.* (2001)] or proposing random permutations. TDMA-based buses have also been proven amenable for MBPTA by padding execution time measurements conveniently [Panic *et al.* (2015)]. A tree NoC implementing wormhole routers with random arbitration and intended for all-to-one communication has also been shown to be amenable for MBPTA has been explained in Chapter 4. However, trees do not fit well all-to-all communication.

Summary. Our work targets achieving time-composable WCET estimates on high-performance wNoC designs for all-to-all communication. To do so, we rely on existing MBPTA randomization techniques. Like [Lu *et al.* (2014), Bogdan *et al.* (2010)] we exploit probabilistic analysis to avoid overdimensioning network contention. However, we introduce modifications (randomization) in the network that make contention to have by construction a probabilistic behavior instead of modeling application traffic probabilistically. Thus, we enable the derivation of WCET estimates with MBPTA by smartly limiting contention.

5.6 Conclusions

In this Chapter we show that appropriate probabilistic approaches are highly efficient dealing with contention in wNoCs. Pathological worst-contention scenarios occur with (provable) negligible probability and hence, there is no need to account for them. We propose two different wNoC setups, **LNR** and **LFR**, that are able to provide much better performance guarantees than deterministic approaches by making use of a wormhole router with randomized arbitration. **LNR** is particu-

larly suitable for scenarios with moderate $\overline{WCD}$ values and for applications that are very sensitive to latency, and **LFR** suits better large NoCs where $\overline{WCD}$ values are expected to be huge.

Chapter 6

Credit-Based Arbitration

AS we have shown in previous chapters, fair arbitration in the access to hardware shared resources is key to obtain low worst-case execution time (WCET) estimates in the context of CRTES. A number of hardware-only mechanisms exist for managing arbitration in those resources (buses, memory controllers, etc.). However, they typically attain fairness in terms of the number of requests each contender can issue to the shared resource. As we show in this Chapter, this may lead to unfair bandwidth allocations when one contender issuing short requests competes with contenders issuing long requests. Therefore, we propose Credit-Based Arbitration (CBA), a new design of a bandwidth allocation mechanism achieving fairness at cycle level rather than at request count level.

6.1 Introduction

As explained in Chapter 2, multicore contention has been shown to be a key performance limiter for safety-related real-time functions if hardware is not designed properly and/or timing analysis is unable to account for its impact reliably and tightly.

Shared buses have been shown to be effective to cope with the bandwidth requirements of small (up to 4) size multicores to reach shared L2 caches and memory [Salminen *et al.* (2007)]. A number of arbitration policies have been proposed to choose what core is granted access to the shared bus and other resources when contention occurs. Among those one can find round-robin, FIFO, TDMA, lottery and random permutations [Kelter *et al.* (2014),Jalle *et al.* (2013b),Lahiri *et al.* (2001),Jalle *et al.* (2014)]. All those policies have been shown to provide a high degree of fairness across cores in terms of number of requests granted. However, whenever requests from different cores have different duration, fairness is lost since cores with larger requests hog most of the bandwidth to the detriment of cores

with shorter requests. For instance, if two cores are granted access alternatively to a shared resource, one of them with 5-cycle requests and the other with 45-cycle requests, the first one only uses 10% of the bandwidth whereas the latter uses 90% of the bandwidth.

In this Chapter we tackle this issue by proposing a credit-based arbitration (CBA) policy that balances shared resource utilization by tracking how long each contender has used the shared resource. CBA provides each contender with a time budget that is decreased by the amount of time the shared resource is used and recovered slowly later to ensure that no contender (core) overuses the shared resource. This way those cores issuing short requests are granted access more often than those issuing long requests, thus achieving bandwidth fairness. We do so in the context of Measurement-Based Probabilistic Timing Analysis (MBPTA) [Cucu-Grosjean *et al.* (2012)], thus proving that reliable and tight WCET estimates can be obtained on top of a multicore implementing CBA.

6.2 Background

Arbitration policies are needed to grant access to shared resources to the different contenders. Existing policies typically provide some form of fairness across contenders. In the context of multicores connected to L2 and/or memory with buses, such fairness is provided across different cores to access the bus and so the other shared resources. Many policies exist for that purpose, but only some of them have been shown to be amenable for safety-related real-time systems where WCET needs to be tightly and reliably estimated for scheduling purposes.

In general, policies like FIFO, round-robin and TDMA can be regarded as fair w.r.t. the number of requests, but not w.r.t. the duration of those requests. For instance, let us assume a scenario where all cores issue requests constantly to the shared bus. In the case of FIFO and round-robin, the different cores will access the bus alternatively keeping it fully utilized. In the case of TDMA, time is typically split across cores homogeneously and it is also common using time slots whose duration matches the longest duration of any request [Jalle *et al.* (2013b)]. Assuming that the duration of a request is unknown a priori (i.e. whether it will hit/miss in L2, whether it will produce a dirty line eviction in L2, etc.), TDMA will typically allow requests to be issued only in the first cycle of the corresponding slot for each core. Allowing a request whose duration is unknown to be issued at any other time could prevent requests from other cores being issued at their expected time, which is not allowed for being able to estimate the WCET. Therefore, in our scenario where all cores issue requests constantly, cores will be granted access to the bus alternatively, but leaving the bus idle whenever a request takes less than the maximum latency.

Figure 6.1: Chronogram showing requests arbitrated with and without CBA.

Other MBPTA-compliant arbitration policies, such as lottery [Lahiri *et al.* (2001)] and random permutations [Jalle *et al.* (2014)], take decisions randomly with different constraints with the aim of making the worst case being closer to the average case and thus, improve WCET estimates. However, in our particular example where the bus is fully congested, in the long run they share the bandwidth to the bus homogeneously across cores *in terms of number of requests*, as it is the case for FIFO, round-robin and TDMA.

Overall, all those policies have a common characteristic: under high congestion, they balance how many times each core is granted access to the bus, but they neglect how long each core uses the bus. For instance, in our particular example with a fully-congested bus and 4 cores, if cores 1, 2 and 3 issue requests whose duration is 50 cycles and core 4 issues 5-cycles requests, cores 1, 2 and 3 will have to wait 105 cycles on average to use the bus during 50 cycles for all those policies (except TDMA where waiting time will be 150 cycles). Therefore, those cores will experience a 3.1x slowdown (4x for TDMA). Instead, core 4 will have to wait 150 cycles to use the bus during 5 cycles (except for TDMA where waiting time will be 195 cycles). Therefore, core 4 will experience a 31x slowdown (40x for TDMA).

In general, one would expect up to 4x slowdown when consolidating in a multicore 4 tasks that fully utilize a shared resource each one in isolation. However, inappropriate design choices may lead to performance issues such as those related to short requests competing with long request, as already shown in some processors [Fernández *et al.* (2012)]. Thus, specific arbitration policies that fairly share bandwidth in terms of time rather than in terms of requests are needed. We propose CBA to cover this gap.

6.3 Credit-Based Arbitration

In this section we introduce, first, an illustrative example of fair request arbitration leading to unfair bandwidth allocation. Then we introduce CBA design and describe its implementation.

6.3.1 Motivation: An Example

We present next an example that relates to the hardware platform used later for the actual implementation of CBA. Let us assume that we have a 4-core processor with a bus to connect cores with a partitioned L2 cache and the memory system, where a request holds the bus until it is fully served. This is referred to as a non-split bus. In this context, let us assume that the task under analysis issues frequent requests that access the L2 cache with a total turnaround latency of 6 cycles once granted access to the bus. On the other hand, tasks in the other cores are streaming applications issuing constantly read requests to memory that take 28 cycles.

Under this setup the bus would be fully saturated by any of the tasks. When consolidating the 4 tasks in the 4 cores so that all of them run simultaneously, any request-fair arbitration policy will lead to a situation where roughly 25% of the requests in the bus belong to each of the cores. Hence, each 6-cycle request of the task under analysis will have to wait for around 28x3=84 cycles to be granted access to the bus. If the task under analysis runs for 10,000 cycles in isolation out of which 6,000 cycles are spent accessing the bus (1,000 requests), its execution time with contention will be easily close to $(10,000 - 6,000) + 1,000 \times (6 + 84) = 94,000$. In other words, in a 4-core processor this task can experience a 9.4x slowdown. If a cycle-fair arbitration was used, execution time would be instead $(10,000 - 6,000) + 1,000 \times (6 + 18) = 28,000$, so a 2.8x slowdown. While such slowdown is still very high, it is expected when tasks saturating a given resource in isolation are consolidated together in the multicore. However, the expectation would be that, given N cores, the slowdown should be at most Nx.

6.3.2 CBA Design

CBA builds upon several premises:

1. Requests will be eventually granted access regardless of their duration, as long as a maximum duration exists and its latency is known (or can be upperbounded). We define this maximum duration (or its upperbound) as $MaxL$.

2. Requests will be eventually granted access regardless of the core they belong.

3. Bandwidth is fairly distributed across contenders in terms of cycle counts rather than in terms of request counts.

This is practically achieved by allocating each core a given credit (budget) matching $MaxL$. Then, arbitration is performed across all cores with pending requests and an available budget of exactly $MaxL$ cycles. When a request is

granted access to the bus, the budget of the corresponding core is decreased by the bus hold time. For instance, this can be implemented by decreasing by 1 the budget of the core using the bus. In parallel, every cycle all cores get their budget increased in the following way:

$$Budget_i(t+1) = \min(Budget_i(t) + 1/N, MaxL) \tag{6.1}$$

where $Budget_i(t)$ stands for the budget of core i in cycle t and N stands for the number of cores. Note that budget saturates at $MaxL$ to prevent the case in which one core spends long time not using the bus and then it hogs the bus during a long time period. Otherwise, the effective bandwidth enjoyed by one task would depend on the shared resource utilization performed by previously executed tasks in all cores, which could lead to any arbitrary budget imbalance. Instead, with our approach we only need to collect measurements at analysis time making the task under analysis (TuA) start with zero budget. Also note that, although conceptually the budget is increased by a fraction, this can be implemented by multiplying all factors in Equation 6.1 by N. In that case, when using the bus, the budget should also be decreased by N every cycle instead of by 1.

CBA operation is illustrated in Figure 6.1 where each core has always requests ready and they are arbitrated in the following order: A, B, C, D, B, A, D, C, D, B, C, A, B, C, A, D, A, C, B, D, A, B, C, D. Requests from cores A, B and C take 28 cycles and from core D, 6 cycles. We focus in the first 336 cycles (the time needed to hypothetically send 3 28-cycles requests from each core). As shown, without CBA only 3 requests from core D are served in 336 cycles (so it used only 5.4% of the bandwidth), but keeping the bus fully utilized. However, with CBA in this timeframe core D gets 7 requests arbitrated instead of only 3. In the case of CBA, whenever the arbiter indicates that the next core is i, i is granted access if it has enough budget. If not, we move in the sequence until we find a core with enough budget, as long as at least one core has enough budget. For instance, after the first request of D is served (cycle 90) no core has $MaxL$ budget yet, so the bus remains idle until at least one core has $MaxL$ budget. At that point, the one recovering its budget earliest is core D (in cycle 108), so in cycle 108 D is granted access and we skip B and A in the sequence (and also D since it is arbitrated).

Note that in the process of guaranteeing higher bandwidth to D, CBA wastes 34 cycles out of 336. On the other hand, the number of D requests arbitrated grows by more than 2x.

6.3.3 Arbitration Choices

CBA acts as a filter to choose what pending requests are eligible to be arbitrated: only those whose core has $MaxL$ budget can be arbitrated. Then, any arbitration

policy can be applied on top. Given the particular timing analysis considered in this Chapter, MBPTA, we could implement on top any of the arbitration policies shown to be compatible with MBPTA: round-robin, lottery, random permutations [Jalle *et al.* (2014)], or TDMA [Panic *et al.* (2015)].

As shown in Equation 6.1, homogeneous bandwidth allocation builds upon increasing the budget of all cores by $1/N$ every cycle, where N stands for the number of cores. Hence, under maximum utilization scenarios, the budget of one core is decreased by 1 and the budget of each of the N cores is increased by $1/N$, so overall budget increases also by 1, thus allowing full bus utilization.

Heterogeneous arbitration across different cores, thus giving higher bandwidth to some cores than to others, would also be possible in several ways as detailed next.

Overbudgeting

One may let the budget of some cores grow above $MaxL$. For instance, we could allow the budget of one of the cores grow to up to $2 \times MaxL$. If this core does not send requests for a while, it could eventually send requests back-to-back, which is good for this core but creates some temporal starvation to the others. In this example, this particular core could send up to two consecutive requests if they have maximum latency, or further requests if they have lower latency so that the remaining budget is equal or higher than $MaxL$. Note, however, that this approach provides higher bandwidth to the cores whose budget can grow above $MaxL$ only if they have not used the bus much in the previous cycles. Instead, if those cores send requests sustainably, they will only reach $1/N$ total bus utilization since this is the speed at which budget is recovered.

Overspeed

An alternative to have heterogeneous bandwidth consists of increasing the budget of all cores (in total) by 1 every cycle but in a heterogeneous way. For instance, we could make bandwidth grow by 1/2 for core 1 and by 1/6 for each of the other 3 cores in a 4-core processor. This would allow core 1 recover budget faster, and so having a 50% total bus utilization. Still, differently to overbudgeting, this approach would not allow core 1 to send requests back-to-back since, after sending a request of duration L cycles, it would have to wait for another L cycles before its budget reaches $MaxL$. Note that budget is recovered also during those cycles when the bus is used, so that budget decreases by 1 cycle and recovers by 1/2 cycles every cycle the bus is used.

Combined

A third alternative consists of a combination of overbudgeting and overspeed, so that the budget of some cores grows at a faster rate than for others, and their maximum budget is above $MaxL$. This would allow those cores taking a larger fraction of the bus utilization and, at the same time, issue requests back-to-back.

6.3.4 WCET Estimation

MBPTA relies on measurements capturing the worst (probabilistic) conditions to deliver bounds that hold under any conditions. As explained in Chapter 2, this can be done by collecting execution times of the TuA under maximum contention. In our case this can be done enforcing the two following conditions:

1. Contending cores always have a request ready to compete with the TuA, but new requests are created only if the TuA has a request ready.

2. Contending requests always have maximum latency, $MaxL$.

The first condition creates the highest contention since the requests of the TuA always find the maximum number of contenders ready. Note that by not creating requests when the TuA has no pending requests, contenders are more likely to have all their budget available by the time the TuA generates a new request. In other words, contenders only consume their budget when their requests compete against those of the TuA. The second condition relates to the fact that, when competing, requests from contenders are greedy creating the highest contention as soon as possible. This holds under the premise that the impact of contention in execution time is the same for different requests of the TuA, which is often the case in simple in-order processors such as those used for safety-related real-time functions [Cobham Gaisler (2017)]. In other words, if at some point contenders do not have enough budget to create contention for a given request of the TuA, it can only be because that budget was used creating contention for an earlier request of the TuA. Finally, as stated before, measurements for the TuA are collected under worst conditions also in terms of its own budget, which is zero, thus delaying the most the issuing of the first request of the TuA. By that time all contenders will also have all their budget available to compete with the TuA.

6.3.5 Implementation

Since implementation complexity and overheads can make industry simply dismiss some solutions due to their unlikely viability, we have implemented CBA in an FPGA design of a 4-core LEON3 processor used in the Space domain [Hernández

Table 6.1: Summary of signals

	Every cycle	**When using bus**
$BUDG_i$	$\min(BUDG_i + 1, 228)$	$BUDG_i - 4$

	WCET mode	**Operation mode**
$COMP_1$	—-	—-
$COMP_{2,3,4}$	$BUDG_i == 228 \wedge REQ_1 == 1$	1
REQ_1	when request ready	when request ready
$REQ_{2,3,4}$	1	when request ready

et al. (2015)]. In particular, we have implemented CBA together with random permutations arbitration as it has been shown to be the MBPTA-compliant policy with best performance [Jalle *et al.* (2014)]. $MaxL$ is 56 cycles since memory latency is 28 cycles and the longest requests may produce 2 memory accesses. For instance, atomic operations produce a read and a write operation. Also, L2 cache misses evicting a dirty line produce 2 operations: one to write dirty data back to memory and one to fetch requested data.

The processor implements a non-split AMBA bus [ARM (1999)]. In general, buses with split transactions have more homogeneous request sizes. However, even in a system with split the worst-case situation, having very long and very short requests, is possible since atomic operations by definition cannot be split. We have implemented CBA as a part of the AMBA bus arbiter and connected it to the APRANDBANK module that delivers random bits every cycle for random choices of the random permutations arbitration [Hernández *et al.* (2015),Agirre *et al.* (2015)].

Next we describe the signals used, which are conveniently summarized in Table 6.1. For each core the arbiter has an 8-bit budget counter ($BUDG_i$) that saturates at 228 (56x4). Every cycle all $BUDG_i$ saturated counters are incremented by 1. Also, every cycle the core using the bus (if any) gets its $BUDG_i$ counter decreased by 4. Our implementation allows configuring the platform as either *WCET estimation* or *operation* mode. During *WCET estimation* mode the request (REQ_i) signals of cores 2, 3 and 4 are always set. Each of those cores has also a compete ($COMP_i$) bit. The $COMP_i$ bit is set when $BUDG_i$ is 228 (so $MaxL$) and REQ_1 is set, thus meaning that the TuA, which runs in core 1, has a request ready. $COMP_i$ is reset whenever core i is granted access to the bus. Also, during *WCET estimation* mode cores 2, 3 and 4 keep the bus busy during 56 cycles when granted access.

During *operation* mode, REQ_i signals are only activated when the corresponding core has a request ready and $COMP_i$ signals are always set, thus not making requests wait for requests in core 1.

6.4 Evaluation

In this section we present the evaluation framework and some results comparing CBA vs no-CBA in the context of a bus implementing random permutations arbitration and using MBPTA to derive the WCET.

6.4.1 Experimental Framework

Multicore Setup. We use a 4-core processor with shared bus and shared L2 memory. It implements pipelined in-order SparcV8 LEON3 cores for the Space domain with random-placement and random-replacement caches, as needed for MBPTA [Hernández *et al.* (2015)]. Data L1 cache implements write-through policy, whereas L2 cache implements write-back policy. Cores are connected to a shared (partitioned) L2 cache through an AMBA bus [ARM (1999)]. L2 cache is connected to a memory controller that serves as bridge to DRAM DDR2 memory. Therefore, bus transactions take between 5 cycles (L2 read cache hit) and 56 cycles (e.g., L2 miss producing a dirty line eviction). We use random permutations policy to take arbitration decisions [Jalle *et al.* (2014)]. This architecture has been prototyped in an ALTERA TerasIC DE4 FPGA.

Multicore Model. For the sake of facilitating the understanding of the impact of CBA, we have also built an analytical multicore model where we interleave computation phases and bus accesses with a fixed period. For instance, we consider a *1% LD L2 hit* setup where the task under analysis executes constantly 99 1-cycle core operations followed by a 5-cycle bus access.

Workloads. Results on the FPGA have been obtained for the EEMBC Autobench suite [Poovey *et al.* (2009)].

Railway Case-study. The mixed-criticality railway case study is comprised of two self-contained subsystems. (1) Simplified European Train Control System (ETCS) railway signaling subsystem: Safety-critical application (Safety Integrity Level (SIL) 4) that protects the train by supervising the traveled distance and speed, activating the brakes if authorized values are exceeded. The software architecture of this subsystem is composed of three tasks executed sequentially: a. Odometry module (OMS): is the responsible of estimating a set of parameters based on the information received from the train environment (e.g., estimated trains position). b. Emergency module (ES): Controls the Emergency braking system. c. Service module (SS): Controls the Service braking system. 2. Traction control subsystem: It controls the speed/pair of the electric motors varying the switching frequency of a power inverter. This subsystem is not safety related but it involves stringent real-time requirements. This subsystem is distributed in two different cores, the first one holds the traction control and the second one a model of a Permanent-Magnet Synchronous Motor (PMSM) for closed-loop control.

MBPTA. The MBPTA technique is applied to a subset of the safety-critical ETCS subsystem. In particular, it is used to estimate the pWCET of the Emergency module (henceforth referred as Unit of Analysis (UoA)). This UoA is composed of several data dependent paths. To conduct the measurement based analysis, a set of input vectors has been defined, which exercises the UoA at basic block level. These input vectors characterize the train environment and are sent to the platform by an Ethernet communication channel. Each input vector triggers a different unique path of the UoA. There are a total of 10 input vectors (that, hence, exercise 10 paths). By default, measurements are taken on a per-path basis, which allows constructing a separate pWCET for each of the traversed paths.

6.4.2 Results

Synthetic. First, we have evaluated the potential gains of CBA with our analytical multicore model. For that purpose we consider that the task under analysis executes constantly a number of 1-cycle core operations followed by 1 bus access. We vary the number of core operations between bus accesses to represent a different fraction of bus accesses. Hence, *1% LD* has 99 core operations between each pair of bus accesses, *2% LD* has 49, *5% LD* has 19 and *10% LD* has 9. For each such fraction we consider 4 cases: either requests have a 5-cycle latency (LD L2 hit) or 28-cycle latency (LD L2 miss without dirty eviction). Results show the slowdown when experiencing contention in a 4-core setup against tasks issuing constantly 28-cycle requests. Results are normalized w.r.t. the execution without any contention.

Results are shown in Figure 6.2. We observe that, whenever requests of the TuA are long (L2 miss), the slowdown in the 4-core processor approaches 4x as the bus utilization increases. In particular, slowdown is 5x for no-CBA with *10% LD* due to the fact that most of the time the TuA is accessing the bus (9 core cycles followed by 28 bus cycles) and sometimes random permutations grant access twice consecutively to the TuA, which misses the second arbitration slot due to the 9-cycle core latency. For instance, if the TuA is granted access last in one arbitration window and first in the following one, it will use its slot in the first arbitration window and its following request will arrive too late to use the first slot in the second window. Thus, the TuA will spend the complete arbitration window waiting. With CBA, instead, when the TuA and its contenders spend most of the time accessing the bus, it is often the case that only one core has budget to access the bus, thus removing the effect of the missed window.

When requests are short (L2 hit), as expected, the TuA experiences increasing slowdowns that exceed 4x by far (up to 9.4x) without CBA since each 5-cycle request competes against 3x28-cycle requests. Conversely, with CBA slowdown is always lower than without CBA (and well below 4x) since contenders take longer

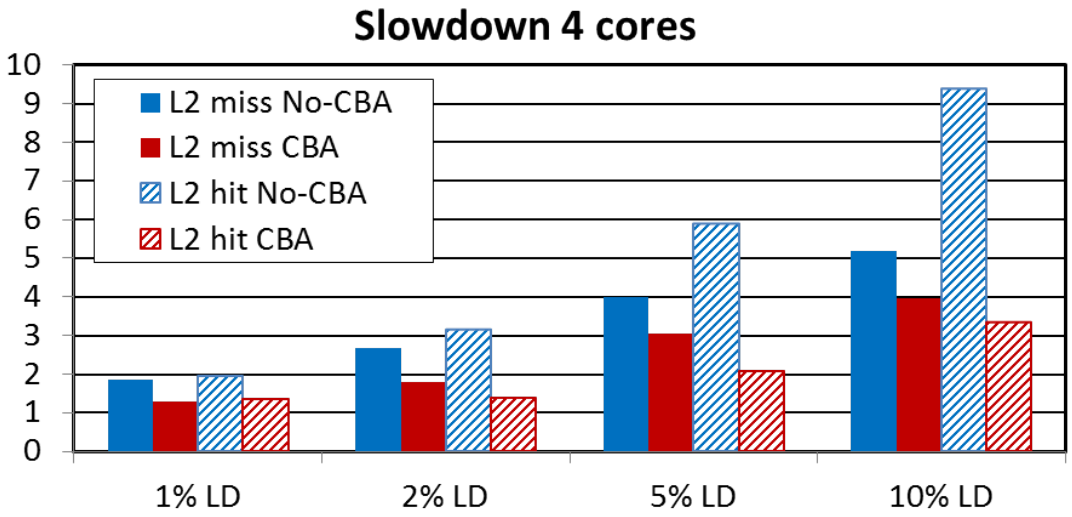


Figure 6.2: Slowdown with and without CBA for different synthetic examples.

to recover their budget than the TuA, which is able to issue requests with higher frequency as illustrated in the example in Figure 6.1.

EEMBC. We have run EEMBC benchmarks on top of the FPGA for 3 different bus configurations, all of them using random permutations to take arbitration decisions. Those configurations include the baseline bus without CBA (RP), the credit-based one (CBA), and a heterogeneous implementation of the CBA (H-CBA) where the TuA gets 50% of the bandwidth. The TuA is run in those 3 configurations in isolation and with maximum contention. Since the bus performance strongly depends on the number of accesses a given benchmark performs to the bus we show average execution time results for 1,000 runs of each configuration. Having a significant number of runs is important to carry out a fair quantification of bus performance since in our platform cache behavior and bus arbitration are randomized.

Results in Figure 6.3 are normalized for every benchmark w.r.t. the result obtained for RP in isolation. As shown, with EEMBC slowdowns are below 4x. This occurs because EEMBC benchmarks do not saturate the bus. Still, we can see that when CBA is not in place, slowdowns can be very significant under maximum contention (3.34X for `matrix`). On the contrary, when CBA is used, execution times of tasks under maximum contention are much lower suffering in the worst case a 2.34% slowdown. We have also evaluated a configuration in which the TuA receives more bandwidth than its contenders (H-CBA). In particular, each cycle the TuA recovers 1/2 cycles of budget and each other core only 1/6 cycles. This virtually allocates 50% of the bandwidth to the core where the TuA runs.

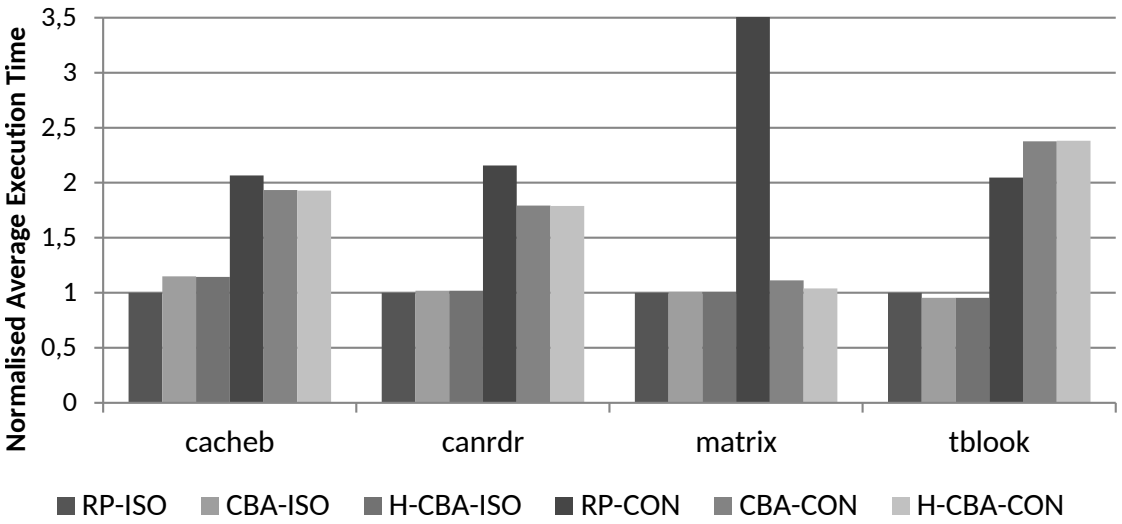


Figure 6.3: Slowdown with and without CBA for EEMBC on the FPGA multicore. ISO stands for isolation and CON for maximum contention.

As shown in the Figure, H-CBA reduces the maximum slowdown experienced across benchmarks, thus showing the effectiveness of CBA also with heterogeneous bandwidth allocation. However, this solution is not very suitable for EEMBC workloads since they do not have significant bandwidth requirements and this configuration would worsen the performance of any task running in the other cores.

Another important effect we can observe is the impact that CBA arbitration has in the execution time of tasks in isolation. Given that under CBA a core is not granted access to the bus until it has enough budget, the execution of a task can get stalled. However, as we can see in Figure 6.3, this effect is not very important and its impact depends on how often a program has a request ready before having recovered its budget. In fact, we observe that CBA increases only the execution time by 3% on average w.r.t. the RP bus in isolation. Moreover, when H-CBA is deployed, the impact is negligible being very close to the RP bus on average. Note that, for `tblook` we observe slightly better performance with CBA in isolation than with RP arbitration and worse performance with contention with CBA than with RP. We have investigated these scenarios and verified that two conditions concur in this case: (1) those benchmarks are almost insensitive to the potential delays created by CBA since their bus requests barely occur consecutively in time; and (2) those benchmarks are highly sensitive to the particular (random) cache placements experienced in the experiments. Therefore, those cache placements influence notably average execution times depending on whether "bad" cache placements occur more or less often. While this may have an effect on average performance,

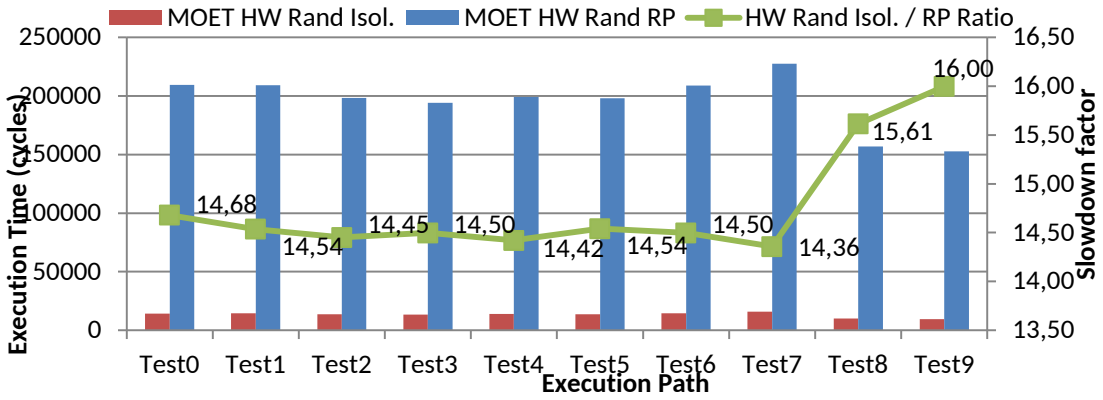


Figure 6.4: Slowdown without CBA for the Railway application.

it does not affect WCET estimates since MBPTA builds upon EVT, which keeps only the group of high execution times to predict the WCET.

Railway Case-study. Figure 6.4 shows the execution times of each path with the random permutation bus arbitration (without CBA) and when running in single-core mode. The green line in the figure below shows the slowdown introduced by the random permutation policy configured with a fixed slot duration of 56 cycles (14x-16x). This slowdown is expected as the case study performs abundant bus accesses due to the abundant store instructions (around 30% of the instructions across tests) and the fact that the DL1 is write-through. Therefore, the bus is highly saturated already when running in isolation, thus making the application highly sensitive to bus contention.

The random permutation arbitration forces each core to wait until its time slot to be granted access to the bus. Slots are grouped in time windows where each window has one slot per core (four in our setup) and the core to slot allocation is done in a random way. This provides fairness in terms of number of requests granted per core. Enforcing the slot size to 56 cycles for all co-runners ensures time composability since the arbitration does not depend on the particular requests raised by the contenders in other cores. However, most requests hit in L2. For instance, Test7 performs up to 1534 L2 accesses in the measurements collected, but only up to 256 L2 misses. Therefore, around 83 % of the bus accesses hit in L2 and thus, experience a short latency. However, those accesses are typically delayed by three 56- cycle requests from the other cores on average to obtain time-composable pWCET estimates. This leads to a scenario where each short request (e.g., 5-8 cycles) is made to wait for up to 168 cycles on average (3x56 cycles).

Figure 6.5 shows the slowdown of CBA (red bar) with respect to results in isolation and compare it with RP (blue bar). We see that the performance has considerably improved, with slowdowns due to multicore contention around 2x

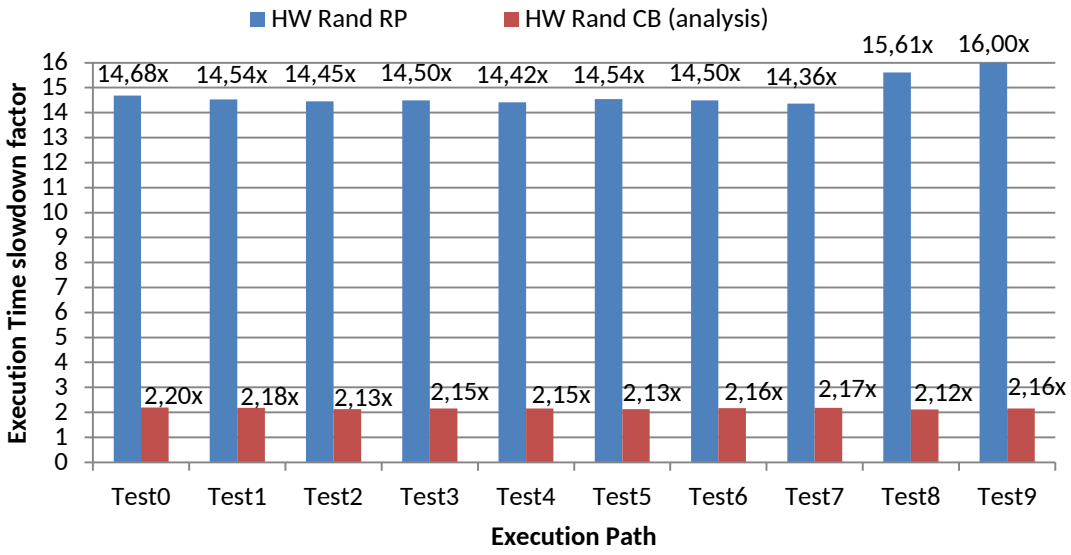


Figure 6.5: Slowdown with and without CBA for the Railway application.

(for RP it was around 15-16x). This is in line with the experiments done with the multicore model. By achieving time-fairness rather than request-count fairness, credit-based arbitration allows moving from a scenario where each request has to wait for more than 100 cycles to be granted access to the bus, to a scenario where each request can proceed experiencing much lower contention due to the much higher availability of the bus when short requests are issued.

Overheads. Hardware overheads of CBA can be regarded as negligible. In particular, the processor model has been synthesized at 100MHz – the maximum operating frequency for our TerasIC board – with and without CBA. Also, the FPGA occupancy without CBA is 73% and it has grown by far less than 0.1% to implement CBA. Hence, our arbitration policy is simple enough to be implemented in real designs.

6.5 Related Work

Literature on Networks-on-Chip (NoCs) for safety-related systems is abundant and includes buses, rings, trees and meshes among others [Jalle *et al.* (2014),Panic *et al.* (2013),Schoeberl *et al.* (2012)]. Some designs are specifically tied to TDMA arbitration, such as Nostrum [Millberg *et al.* (2004)] and Aethereal [Goossens *et al.* (2005)], and require software layers to schedule at very fine granularity communications due to the use of contention-free packet transmissions [Bui & Lee (2012)]. This is generally unaffordable when cache memories are used for

performance purposes due to the difficulties to schedule, for instance, cache misses. A number of works have focused on providing tight bounds for the worst-case traversal time in complex NoCs such as meshes [Rahmati *et al.* (2013),Psarras *et al.* (2015),Wassel *et al.* (2013),Panic *et al.* (2016b)].

However, safety-related real-time systems have serious difficulties to deliver evidence needed for certification purposes even for very small multicores. Most such systems still focus on single-core processors and only small multicores have been realistically considered. For instance, dual-core processors have been accepted under a number of limitations in avionics [Federal Aviation Administration (FAA) (2014),Cullmann *et al.* (2010)]. Analogously, microcontrollers with 2 or 3 cores have been successfully accepted in the automotive domain [Infineon (2012),Cullmann *et al.* (2010)]. In those cases, simple interconnection networks such as buses or crossbars have been accepted, thus being the arbitration centralized either in the interconnection or in the target device (e.g., memory).

Arbitration policies for hardware shared resources in the context of safety-related real-time systems are also abundant as described in Section 6.2. However, most policies have been implemented pursuing fairness across request counts from the different contenders. The fact that requests may have different durations makes that request-fair policies fail to be fair in terms of actual utilization of the shared resource across contenders. CBA tackles this issue with specific means to prevent contenders with long requests from hogging the shared resource, thus allowing a more balanced use of the shared resource across contenders.

6.6 Conclusions

Existing policies to arbitrate the access to hardware shared resources in multicores focus mostly on achieving fairness in terms of request counts rather than in terms of time. This leads to severe slowdowns for tasks issuing frequent-but-short requests to shared resources.

In this Chapter we propose CBA, an arbitration policy that allows a fair sharing of hardware resources by balancing the true utilization of those resources. Moreover, we show that implementation costs of CBA are affordable by implementing it in a Space multicore prototyped in a FPGA. Our results show that the maximum slowdown roughly matches the core count – as one would expect – when all tasks saturate the shared resource, which compares to existing policies whose slowdown is virtually unbounded.

Chapter 7

Eviction Frequency Limiting (EFL) for Shared Caches

Shared caches in multicores challenge Worst-Case Execution Time (WCET) estimation due to inter-task interferences. Hardware and software cache partitioning address this issue although they complicate data sharing among tasks and the Operating System (OS) task scheduling and migration. In this Chapter we propose a new hardware mechanism to control inter-task interferences in shared time-randomised caches without the need of any hardware or software partitioning. Our proposed mechanism effectively bounds inter-task interferences by limiting the cache eviction frequency of each task, while providing tighter WCET estimates than cache partitioning algorithms.

7.1 Introduction

Many existing processors in the real-time domain comprise one shared, usually last-level, cache (LLC), like the ARM Cortex A9 and A15, the Freescale P4080 and the Aeroflex Gaisler NGMP. While LLC offers high potential for average performance improvement, it challenges worst-case execution time (WCET) estimation, which has made LLC to be studied in the last years by the real-time community [Liedtke *et al.* (1997),Mueller (1995),Kim *et al.* (2013),Ward *et al.* (2013),Paolieri *et al.* (2009a)].

As explained in Chapter 2, two main cache design 'paradigms' can be found in the literature: conventional timing analysis techniques (either static or measurement-based) [Wilhelm *et al.* (2008)] usually rely on caches that are deterministic in their temporal behaviour (e.g., caches deploying modulo placement and LRU replacement). Meanwhile, Probabilistic Timing Analysis (PTA) techniques [Cucu-Grosjean *et al.* (2012),Cazorla *et al.* (2013),Altmeyer & Davis (2014)] rely on caches

with a *time-randomised behaviour in which hit and miss events have an associated probability for every cache access*. The main difference between time-deterministic (TD) and time-randomised (TR) caches, is that in a TD cache each memory address is mapped into a fixed cache set and way. That is, certain bits of the address (called the index) *determine* the cache set in which the address is mapped, while the way is determined by the replacement policy. In TR caches, however, each address can be mapped to any set (randomly chosen on each execution) and way (randomly chosen on every eviction), since random replacement and placement policies are used [Kosmidis *et al.* (2013a)].

Software cache partitioning [Liedtke *et al.* (1997),Mueller (1995),Kim *et al.* (2013),Ward *et al.* (2013)] and hardware cache partitioning [Paolieri *et al.* (2009a)] have been used so far to control inter-task interaction in the LLC. The former, which can only be used together with TD caches, maps data/code of each task in non-consecutive memory locations so that data/code are mapped into the desired cache sets. This solution, however, may introduce fragmentation in the use of memory and may require significant changes in the memory management. The latter, which can be used together with both TD and TR caches, relies on deploying hardware support to force each task to use a subset of the ways of set-associative caches. In this manner, tasks can be mapped to different ways preventing their interaction. Both solutions are affected by the fact that tasks may have shared pages or libraries, since cache partitioning poses a number of difficulties to allow tasks share data on-chip. Task scheduling is also affected by both hardware and software cache partitioning. In the case of hardware partitioning, partition flushing is required to keep consistency when tasks do not always use the same partition. In the case of software partitioning, two tasks using the same partition cannot be run simultaneously.

In this Chapter we overcome the limitations of cache partitioning by enabling the estimation of trustworthy and tight WCET estimates for systems equipped with fully-shared (non-partitioned) LLCs. The principle behind our proposal is that, while in a TD LLC interferences depend on when (time) and where (the particular cache set in which) misses occur, a TR LLC cache removes any dependence on the particular address accessed and its assigned cache set. This makes that the LLC interferences that a task suffers only depend on how often (frequency) its co-runner tasks miss in cache and not the particular address generating the miss. As a result, *in a TR LLC controlling eviction frequency, by delaying when misses are served for each task, is enough to trustworthily upper-bound the maximum effect that such task may have on other co-running ones in the LLC*. That is, the WCET estimated for a task τ_i is trustworthy regardless of its particular co-runner tasks as long as their aggregated miss frequency – and so their eviction frequency – is below the predefined *miss frequency threshold*, MT_i, for which τ_i's WCET estimate is

computed. Based on this analysis we propose a simple hardware mechanism that limits the miss frequency of tasks in each core at analysis and operation time in a manner that probabilistic upper-bounds can be obtained for the effect in the LLC of one task on the other co-running tasks. Our approach removes cache partitioning constraints while making WCET estimates tighter. This increases the average and guaranteed performance that can be obtained.

7.2 Background on Controlling Cache Inter-task Interferences

Classifying cache accesses as hits and misses in non-shared caches has been already deemed as a complex process subject to some degree of pessimism for those accesses for which it is hard to determine whether they will hit or miss [Mueller (1994),Ferdinand & Wilhelm (1999),Lesage *et al.* (2009),Hardy & Puaut (2008),Reineke *et al.* (2007),Theiling *et al.* (2000)]. Thus, extending this process to the coordinated analysis of several tasks simultaneously sharing a LLC is even harder [Chattopadhyay *et al.* (2010),Zhang & Yan (2012)]. This is so because any shift in the execution time of any task (e.g., due to accesses whose outcome cannot be accurately predicted) would lead quickly to highly pessimistic assumptions for the other tasks, that must account for all potential time alignments across all tasks. Moreover, combined WCET analysis breaks time composability since any change in the task scheduling or the upgrade of any software component invalidates the WCET estimates of all tasks.

Cache partitioning removes the need for multitask cache analysis by ensuring that tasks are assigned a distinct cache portion. Software cache partitioning is done through memory coloring [Liedtke *et al.* (1997),Mueller (1995),Kim *et al.* (2013),Ward *et al.* (2013)]. By enforcing programs' data/code to be allocated in certain memory addresses (pages) it can be controlled the cache sets in which they are allocated. Hardware cache partitioning assigns different cache ways or cache banks to the different co-running tasks such in a way that contention is prevented [Paolieri *et al.* (2009a)].

Cache partitioning complicates task scheduling and data sharing. For instance, let us assume a 4-core processor deploying LLC and executing three tasks, τ_A, τ_B and τ_C. Further assume that, the code and data of each task are mapped in memory such in a way that τ_A and τ_B are mapped on the same cache sets and τ_C is mapped to different cache sets. Under this scenario we observe the following problems with software cache partitioning.

- The scheduler has to prevent τ_A and τ_B from running simultaneously because they would interfere each other in cache.

- It is non-obvious how to manage read-write shared data among τ_A and τ_C since data cannot be simply replicated in cache without challenging functional correctness.

Hardware cache partitioning experiences similar problems for data sharing, but different ones for scheduling. Scheduling problems arise when a task, τ_A, uses a given LLC partition, $Part_i$. Some dirty cache lines may remain in $Part_i$ after τ_A is scheduled out. Whenever τ_A is run again, either it is assigned $Part_i$ or it is given a different cache partition. In the latter case $Part_i$ must be flushed before τ_A is scheduled in for consistency.

7.3 Probabilistically Controlling Eviction Frequency in a TR LLC

7.3.1 Inter-task Interferences in a TD LLC

Tasks can interfere with each other in non-obvious ways in a shared TD LLC. The main features of two co-running tasks, τ_A and τ_X, that shape their interferences in cache are:

1. Their memory mapping (i.e. the memory addresses where their code/data are mapped), which determines the cache sets τ_A and τ_X use

2. The miss frequency in the data cache and the instruction cache of both tasks. The higher the miss rate of a task in the data and instruction caches (assuming a two-level cache hierarchy), the higher its access frequency to LLC and the higher the chances it changes the LLC state and evicts other tasks' data in the LLC

3. Their accesses interleave, that is, the particular order in which accesses occur to the LLC which affects τ_A and τ_X behaviour in cache.

Example The example in Figure 7.1 shows four scenarios illustrating the effect of those three features of two corunning tasks τ_A and τ_X that affect their LLC cache behavior.

Without loss of generality we focus this example on a dual-core processor deploying a two-level cache hierarchy. The first-level private instruction and data caches are deployed per core. For the purpose of this example, we disregard the effect of the interconnection between the cores and the shared LLC (e.g., a full crossbar is in place). The LLC is a 2-set 4-way set-associative cache that deploys modulo placement and LRU replacement. We focus on the task running in $core_0$, which we call τ_A. In the second core we assume a cache hungry task, τ_X, is run.

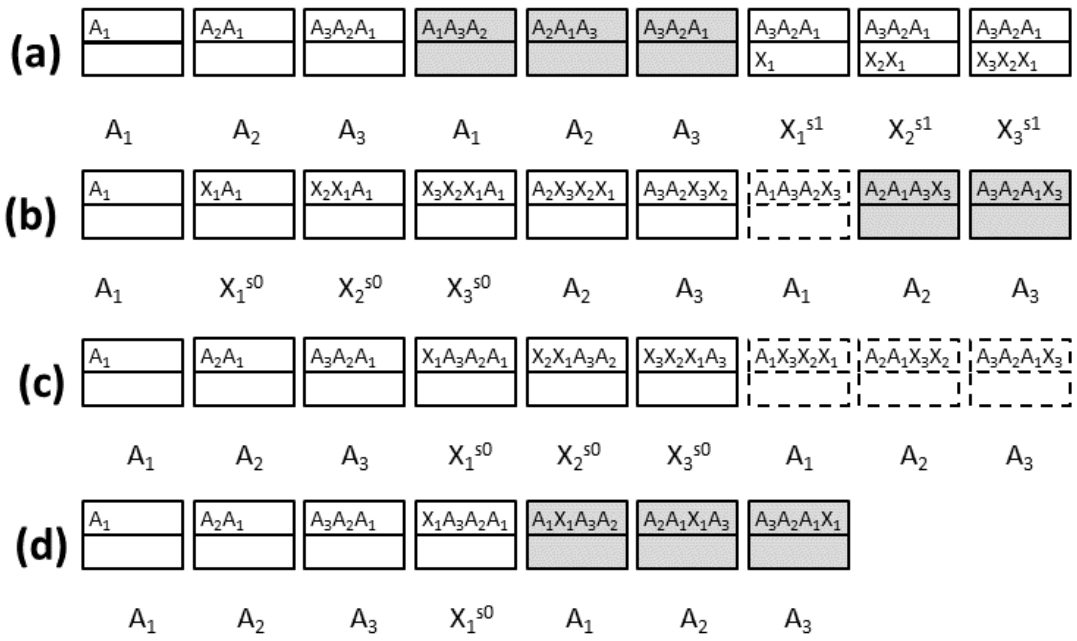


Figure 7.1: LLC state after the sequence of access (6 accesses of core A and 3 accesses of core X). Dotted lines represent a miss after the corresponding access due to inter-task interferences. Grey boxes represent an access hit.

In this example, τ_A (or simply A) makes 6 accesses to LLC which are mapped to set s_0, see Figure 7.1.

- In scenario (a) after τ_A makes its 6 accesses, τ_X makes three accesses that are mapped to set s_1, hence not causing any eviction of τ_A data, which causes no extra misses w.r.t τ_A's execution in isolation
- In scenario (b), all τ_X's accesses are mapped to set s_0 and occur between the first and second τ_A's accesses, causing only one extra miss due to the eviction of A_1, marked as a dotted box in Figure 7.1.
- Scenario (c) is similar to scenario (b), but with τ_X's accesses occurring between the third and fourth τ_A's accesses. This results in the worst-case situation in which τ_X introduces 3 extra misses in τ_A since A_1, A_2 and A_3 are evicted.
- Finally, scenario (d) shows the same situation as in (c) but in this case τ_X only makes 1 access to the LLC, so no eviction occurs.

It is challenging to control, either by hardware or software means, all three factors due to the complex interactions among tasks sharing a LLC. In this example, LLC partitioning, either software or hardware (e.g. to allow task τ_A exclusively set s_0), is the most feasible way to enable the use of a shared TD LLC in real-time multicore systems. However, as explained before, partitioning challenges task scheduling, data sharing and task migration.

This simple example shows that the particular set in which tasks' data are mapped, the order (interleave) of the accesses, and the number of misses experienced lead to different inter-task interference scenarios, thus impacting task LLC behaviour, average and worst execution time.

7.3.2 TR caches

As explained in Chapter 2, TR caches [Kosmidis *et al.* (2013a)] randomise the behaviour of the replacement and placement policies. The placement policy selects, based on some bits of the address, the set to access for a given address; while the replacement policy selects, in the event of a miss in a given set, the victim to be evicted. We use *Evict-On-Miss* (EoM) random replacement, which on the event of a miss, randomly selects a victim line from the target set to be evicted, making it analysable with MBPTA. The random placement policy described in [Kosmidis *et al.* (2013a)] deploys a parametric hash function that uses as inputs the address accessed and a random number, called random index identifier or *RII*. Given a memory address and a RII, the hash function provides a unique and constant cache set (mapping) for the address along the execution. If the RII changes, the cache set in which the address is mapped changes as well, so cache contents must be flushed for consistency purposes. If the RII changes across program execution boundaries, programs can be analysed with end-to-end runs simply by assuming that the cache is initially empty. The hash function proposed in [Kosmidis *et al.* (2013a)] ensures that given a memory address and a set of RIIs, the probability of mapping such address to any particular cache set is the same.

In a TR EoM cache with S sets and W ways, hit and miss events are probabilistic. Given the sequence of accesses $seq_1 =< A_i, B_1, ..., B_k, A_j >$, starting from an empty cache state, with A_i and A_j accessing the same cache line, with no B_l (where $1 \leq l \leq k$) accessing the same cache line as A_i and each B_l accessing a different cache line, the miss probability of A_j can be approximated as [Kosmidis *et al.* (2013a)]:

$$P_{miss_{A_j}}(S,W) = \left(1 - \left(\frac{W-1}{W}\right)^{\sum_{l=1}^{l=k} P_{miss_{B_l}}}\right) \cdot \left(1 - \left(\frac{S-1}{S}\right)^{k}\right) \tag{7.1}$$

The first element in Equation 7.1 is the probability of miss in a fully-associative

cache with W ways deploying EoM random replacement. The base $\left(\frac{W-1}{W}\right)$ is the probability of A_j not to be evicted when a single eviction is performed. The exponent is the addition of the miss probabilities of the elements in between the two accesses to A, which gives a measure of the number of evictions. This first element represents the exact miss probability for a fully-associative cache under the conditions presented above. It becomes an approximation when those conditions change, e.g. when B_l accesses repeat and/or the initial cache state is not empty. However, this is irrelevant for MBPTA, since what really matters is that each access has a probability of hit/miss rather than the particular value of that probability.

The second element in Equation 7.1 approximates the probability of miss in a direct-mapped cache with S sets. Given that placement and replacement work independently, the probability of miss in a set-associative cache with S sets and W ways deploying both random placement and replacement can be computed as the product of these probabilities.

Equation 7.1 provides an intuition on the fact that in a TR cache the key factors affecting the probability of hit of an address A is the number of different accesses carried out in between its current (A_j) and previous access (A_i), called reuse distance of A_j, and the miss probabilities of those accesses.

7.3.3 Inter-task interferences in a TR LLC

As identified in the previous section there are several features affecting the timing behaviour of a task in a TD LLC. Next, we present each of those features for a TR LLC.

Data/Code memory mapping: Under MBPTA the software unit under study is executed enough times according to MBPTA's convergence criteria to ensure representativity of the results. The number of runs required for different benchmarks [Cucu-Grosjean *et al.* (2012),Kosmidis *et al.* (2013a)] and an avionics case study [Wartel *et al.* (2013)] ranges between 300 and 1,000. In each run, a new RII is generated, which makes random placement map each address to a new randomly chosen cache set. This removes the dependence between the memory address of an access and the cache set in which it is mapped.

Access and miss frequency: In a TD LLC hit accesses modify the *LRU stack* (i.e. the bits used by LRU to determine which line is to be evicted in the event of a miss). Interestingly, *an EoM random-replacement policy is stateless* and hits do not alter the cache state. With EoM, on the event of a hit, neither cache (data) contents nor any replacement information are changed. Only misses, which create evictions, alter the cache state. Hence, if $\{B_l\}$ accesses were generated by the co-runners of a given task τ_A's, we would observe that the larger the number (frequency) of $\{B_l\}$, assuming that each access has a non-null probability of miss, the lower the hit probability of τ_A's following accesses (A_j in this case).

Interleave: the particular instant in which co-runners of a task τ_A access the LLC determines which accesses of τ_A experience a decrease in their hit probability, hence affecting τ_A's execution time behaviour.

7.4 Probabilistically upper-bounding inter-task interference features in a TR LLC

In order to derive trustworthy and tight pWCET estimates in the context of PTA by upper-bounding the effect of inter-task interferences in the LLC while preserving time composability, we propose a LLC *eviction frequency limiting* mechanism (*EFL* for short). *EFL* limits how often a task can evict LLC cache lines. To that end *EFL* controls LLC miss frequency. Miss frequency bounds are applied in a different manner during analysis and operation stages such that the timing behaviour observed for a program at analysis time upper-bounds its operation-time behaviour.

Controlling eviction frequency at analysis time. The task under analysis (τ_A) is run in isolation in one core limiting how often it can evict lines from the LLC. *EFL* prevents τ_A from performing an eviction in the LLC until at least Minimum Inter-eviction Delay (*MID*) cycles have elapsed since τ_A last evicted a LLC line. However, LLC hits are allowed to proceed since they do not change LLC state in TR Evict-on-Miss caches. Later we provide details on the hardware implementation. The other cores, by means of our proposed hardware support, generate *artificial requests* that cause LLC evictions at the maximum allowed frequency, i.e. once very *MID* cycles[1]. In this way, τ_A execution times are obtained under the worst intertask interference scenario: maximum eviction rate from other cores.

Controlling eviction frequency at operation time. Each task is allowed, at operation time, to generate a new eviction in the LLC as long as at least *MID* cycles have elapsed since its last eviction. This is enforced by *EFL* hardware (described below). In the worst case, at operation time all the co-runners of τ_A can systematically miss in cache. In order to preserve *time composability*, *EFL* makes no assumption on the miss rate of co-runners. Hence, during analysis time *EFL* forces all the artificial requests to produce evictions in the LLC.

Next we review how inter-task interference features are taken into account by *EFL* .

Data/Code memory mapping. Random placement caches remove the dependence between an addresses and the particular set in which that address is mapped. Hence, the memory addresses in which a program maps its data and

[1]In reality, as explained later in this section, misses occur *on average* every *MID* cycles, but we consider *exactly MID* cycles at this point for the sake of clarity in the explanations.

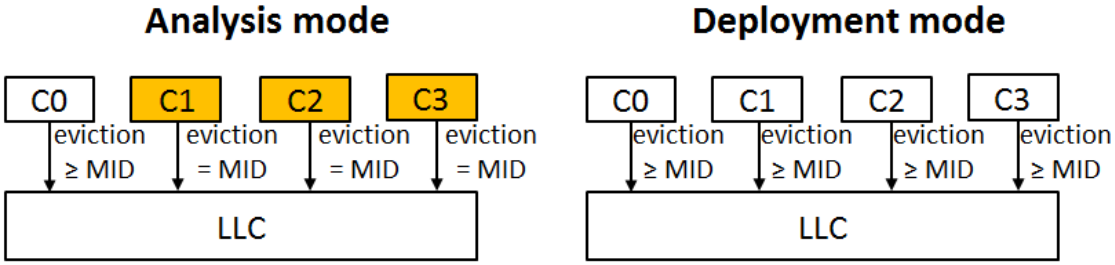


Figure 7.2: Operation mode for each core at analysis and operation time. The task under analysis is run in core 0 (C0) at analysis time.

code affects execution time in a way that is naturally captured by MBPTA [Kosmidis *et al.* (2013a)].

Miss probability. In an Evict-on-Miss TR cache, the higher the probability of miss of the interfering accesses, the higher the probability they evict data of the program under study, τ_A, because on every miss each interfering access causes a LLC eviction. Thus, by enforcing all interfering accesses to cause a LLC eviction at analysis time, our approach upper-bounds the miss probability of the accesses of any co-runner at operation time. This is illustrated in Figure 7.2, where we show how co-runners always perform evictions at analysis time, whereas they access normally at operation time, hence not necessarily missing in every access.

Miss frequency. Our approach imposes a miss frequency at analysis time that cannot be exceeded at operation time. This is so because misses occur *exactly* every *MID* cycles at analysis time, whereas they occur *at most* every *MID* cycles at operation time.

Interleave. How accesses of different tasks interleave may impact tasks' timing behaviour, as explained before. Hence, imposing fixed intervals between evictions could cause systematic effects depending on how those evictions interleave with τ_A accesses. Our approach to avoid such effect consists in randomising how accesses interleave, so that the effect of interleaving can be bounded probabilistically. Under MBPTA, by capturing in end-to-end execution time observations enough outcomes of this randomised event, pWCET estimates capture the effect of such event. Hence, we proceed by enforcing *MID* not to be a deterministic value but instead a random value. If, for instance, the desired *MID* is 1,000 cycles, on every access we set up a new *MID* randomly picked between 0 and $2 \times MID = 2,000$ cycles. By doing so (1) actual *MID* values match, on average, the desired *MID* value. (2) At analysis time the probability of experiencing an interfering access in each cycle is homogeneous and random ($\frac{1}{2 \times MID+1}$). Therefore, interfering accesses interleave randomly. And (3) at operation time the number of interfering accesses between two particular accesses of τ_A is still probabilistically lower than

those experienced at analysis time.

7.4.1 Hardware support

EFL deploys a simple access control unit, see Figure 7.3, that serves as a bridge among each core and the LLC. Through a *rMID* register the system software (i.e. the Operating System) can establish the *MID* value for each core. By means of a *rmode* register, the system software establishes the operation mode, which is either analysis-time or operation-time mode. The Access Control Unit also has a *cache request generator* (CRG) per core, that sends eviction requests to the LLC as required. To that end cache requests (accesses) are flagged: eviction requests created at analysis time by the CRG are flagged with a *force-miss bit*. Such bit is reset at operation time, when the CRG is off [1].

In addition to the *rMID*, *rmode* and the CRG, *EFL* needs to limit LLC miss frequency from each core, both at analysis and operation time. To that end *EFL* deploys a count-down counter (*cdc*) and a pseudo-random number generator (PRNG) per core, see Figure 7.3. The particular PRNG we have used in this Chapter is the Multiply-With-Carry (MWC) [Marsaglia & Zaman (1991)] PRNG, since we have tested that (i) it generates numbers with a sufficiently high level or randomness, (ii) its period is huge, and (iii) it can be efficiently implemented in hardware[2].

On a LLC miss, the PRNG produces a random value in the range $[0, 2 \cdot MID_{desired}]$, which is used to initialise the counter. Such counter is decremented by 1 in each cycle until it reaches zero. *Note that other intervals and probability functions for the latencies are allowed as long as they are kept the same at analysis and operation phases. Keeping the probability distribution function ensures representativeness of the collected execution times during analysis time w.r.t those that may be exercised at operation time.*

The port of the core through which the LLC is reached — either directly or through a bus — is extended with a *eviction allowed (EAB) bit* per core. The EAB is set to 1 when the *cdc* reaches zero. If the EAB is 0, it means that such core is not allowed to perform any eviction. The LLC needs to be enhanced such that on a miss of a request with the EAB set to 0 the eviction is delayed until the EAB is set to 1 and the port for such core is set to busy to block any further access. This allows LLC hits to proceed regardless of the count-down counter contents, but stalls misses, which would cause an early eviction otherwise.

At analysis time, the *rmode* register is set such that CRGs in all cores but the core where the task under analysis (τ_A) runs, issue uninterruptedly eviction

[1] Note that many current ISAs contain in the opcode for load operations 'hint bits' that could be used for EFL purposes.

[2] The access control unit can use the PRNG used to implement random replacement in first level caches as it provides up to 32 bits per cycle, largely above the bandwidth needed.

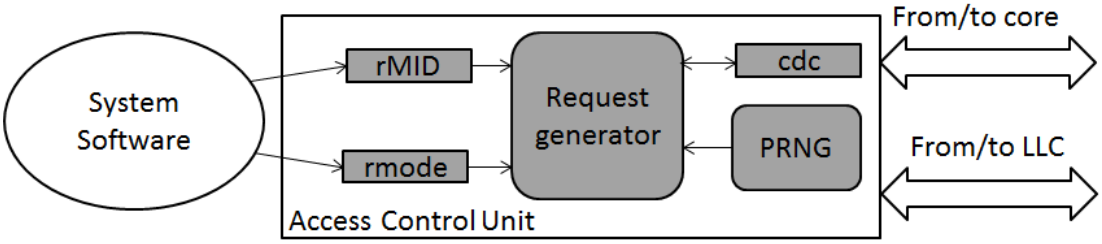


Figure 7.3: Block Diagram of the Access Control Unit

requests to the LLC. All requests, those generated by τ_A and the artificial ones, are controlled by *EFL*, which is also used in all cores at operation time.

While different cores can update the RII of each of their local caches independently, this is not the case for the LLC. Updating the RII of the LLC must occur coordinately at program execution boundaries. This is doable in many domains such as avionics and automotive which rely on Integrated Modular Avionics (IMA) [ARINC (1997)] and AUTOSAR [AUTOSAR (2006)] respectively. Both pursue temporal and spatial partitioning. Temporal partitioning is achieved by splitting execution time into fixed-size time frames, which determine the time budget available for task scheduling. For instance, the incarnation of those time frames in IMA are MInor Frames (MIF) and MAjor Frames (MAF). MIF duration is in the order of few milliseconds. Therefore, the OS can easily change the RII of the LLC at MIF boundaries, which occur coordinately across all cores.

Overall, the hardware overhead is negligible since each core needs a small counter (e.g., a 12-bit counter if $MID_{desired}$ is 2,048) and all cores but one need little logic to generate eviction requests. Regarding the PRNG, since first level caches for data (DL1) and instructions (IL1) already implement random replacement in each core, those PRNG can be easily reused. For instance, the PRNG used in [Kosmidis *et al.* (2013a)] is able to provide 32 random bits per cycle, which are far more bits than needed.

7.5 Evaluation

7.5.1 Experimental Setup

We use a cycle-accurate simulator [Pouillon *et al.* (2009)], as explained in Section 3, to model a 4-core processor. The memory hierarchy is composed of per-core first level separated instruction (IL1) and data (DL1) caches, a shared cache (LLC) across all cores (for both data and instructions) and main memory. IL1 and DL1 are as follows: 4KB, 4-way and 16B-line size implementing random placement and

random replacement (EoM) policies [Kosmidis *et al.* (2013a)]. The LLC is identical to IL1 and DL1 except because its size is 64KB and its associativity is 8. The LLC is non-inclusive. All caches are write-back[1]. Instruction/data accesses have 1-cycle IL1/DL1 hit latency, 10-cycle LLC hit latency and 100-cycle memory latency. The remaining operations have a fixed execution latency in the execution stage (e.g. integer additions take 1 cycle). For the bus, whose access latency is 2 cycles, we use random arbitration policies [Jalle *et al.* (2014)]. For the memory controller we use the solution proposed in [Paolieri *et al.* (2009b)], which upper-bounds the effect of inter-task interferences on the requests of a core to the memory controller.

We use EEMBC Autobench suite [Poovey *et al.* (2009)]. WCET estimation is performed in isolation under the analysis operation mode. For the purpose of measuring average and guaranteed performance we have run 1,024 4-benchmark workloads composed of randomly selected Autobench benchmarks. We collected at most 1,000 measurements (i.e. runs) for each experiment.

7.5.2 Experimental Results

MBPTA compliance

MBPTA compliance of our EFL proposal can be done by probabilistic (a priori) means or by statistical (empirical means).

In order to derive a probabilistic argument about how *EFL* technique is MBPTA compliant we analyse the latencies that a core operation (e.g. integer addition) can take in a multicore processor when deploying *EFL* . Note that core operations may miss in the instruction cache and access the LLC once. Memory operations have similar behavior but may access twice to the LLC, one if they miss data cache and one if they miss the instruction cache, which requires doing twice the analysis we show next.

MBPTA, requires the hardware to guarantee that each operation has its own ETP; however, unlike SPTA, which needs to know all ETPs, MBPTA only requires those ETPs to exist.

We start explaining the ETP of a multicore architecture like the one described in Section 4 when *EFL* is not deployed. As a second step, we will determine how the ETP is affected when *EFL* is active. If *EFL* is not deployed, ETP of a core operation can be derived as follows.

In our core architecture the core latency of instructions, i.e. the latency it takes to fetch and decode is fixed, lat_{core}. Accessing $IL1$ takes also a fixed latency, lat_{IL1}. Whether it hits/misses in the $IL1$ is given with a probability, P_{hit}^{IL1}, see

[1] If a write-through DL1 cache were used, LLC accesses would be much more frequent due to store instructions. In such case, either write operations are not allowed to allocate data in the LLC on a miss or stalls may be frequent with EFL, thus harming WCET estimates and average performance.

Equation 7.1. In the case of a miss, the latency it takes to access to the bus is upper-bounded by a given value lat_{bus}, since we deploy TDMA. Once the memory request gets to the $L2$, it takes a fix latency to access it, lat_{L2}. The probability it hits in $L2$ is given by P_{hit}^{L2} [Kosmidis *et al.* (2013b)]. In the case it misses, it accesses to memory through a shared memory controller in which tasks may suffer inter-task interferences. However, the technique deployed in [Paolieri *et al.* (2009b)] allows upper-bounding the effect of intertask interferences, bound that is introduced at analysis time as lat_{mem}. With all this we have:

$$\begin{aligned} ETP_{coreop} = \{ \quad & \{lat_{core} + lat_{IL1}, \\ & lat_{core} + lat_{IL1} + lat_{bus} + lat_{L2}, \\ & lat_{core} + lat_{IL1} + lat_{bus} + lat_{L2} + lat_{mem}\}, \\ & \{P_{hit}^{IL1}, \\ & P_{miss}^{IL1} \times P_{hit}^{L2}, \\ & P_{miss}^{IL1} \times P_{miss}^{L2}\} \\ \} & \end{aligned} \tag{7.2}$$

Impact of EFL on ETPs. Once we have a miss in $L2$ (LLC) cache, it may be delayed by our *EFL* hardware depending on the time elapsed since the last miss request was sent to the LLC. In particular, the first miss of the program is not delayed at all (we assume $rMID_1 = 0$ before the program starts). Similarly, $dl_1 = 0$.

Whether a particular miss request i is delayed or not depends on (1) the time elapsed since the previous miss in the LLC (dl_i) and (2) the particular (random) delay imposed after such request was issued, $rMID_{i-1}$.

- The probability of each particular value we get in $rMID$ in range $[0, 2 \cdot MID]$ is uniform, $p_{EFL} = \frac{1}{(2 \cdot MID+1)}$. It is important to notice that $rMID_i$ has the same probability distribution at analysis and at operation mode, and by providing uniform distribution it is independent of the particular cycle in which LLC access is granted (no particular systematic alignment with other events can occur).

- The time elapsed since the last miss in the LLC, dl_i, depends on the number of instructions executed between miss request i and the particular miss request $i-1$. Deriving the actual probability of previous LLC cache request is complex, in many cases it is necessary to create the probability tree of all events [Abella *et al.* (2013)], as it is illustrated in the next example. However, to apply MBPTA those probabilities do not need to be known, it is only needed that events affecting the behavior of instructions have a probabilistic

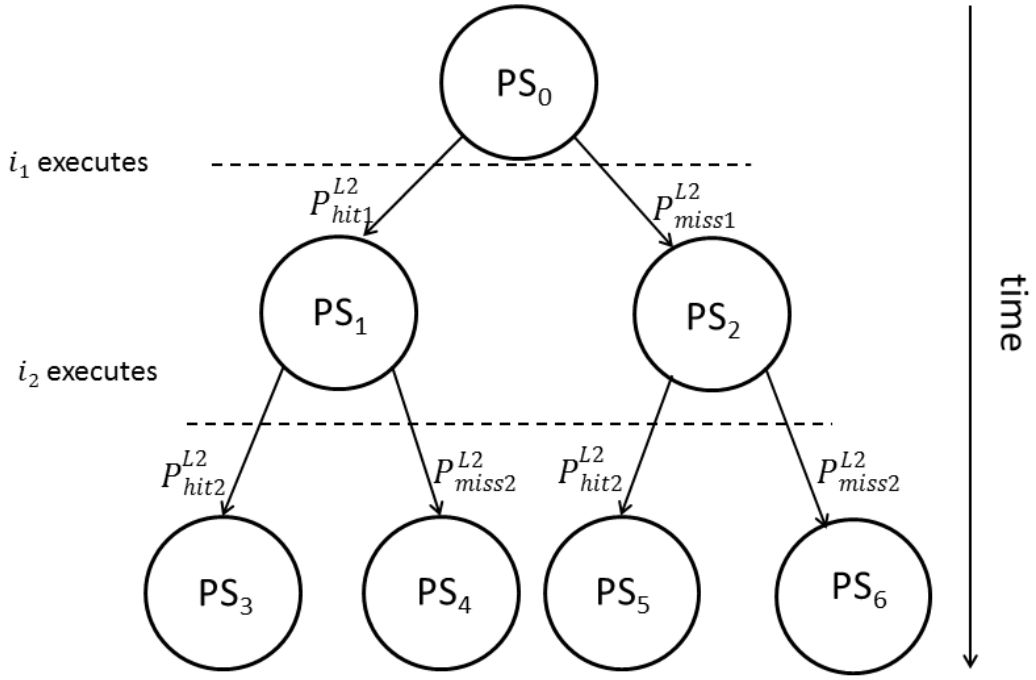


Figure 7.4: probability tree for the two instruction sequence

behavior, which is the case for dl_i and $rMID_i$ (shown below), making our *EFL* design compliant with MBPTA requirements.

Example: Let's assume we have a program comprised of 3 consecutive core operations i_1, i_2 and i_3, and that the LLC $(L2)$ cache has already some data. Further we assume that all instructions miss in the IL_1. Figure 7.4 shows the *probability tree* for the sequence of first two instructions. Each node represents a probabilistic state of the processor, while edges represent the outcome of the random event access to the LLC.

We start from an initial state, in which we assume that $rMID_1 = 0$ and it does not affect the timing behaviour of the first instruction.

The random event 'i_1 access to L2' generates two states, the first one with a probability of $phit_1^{L2}$, called PS_1, and the counter state that happens with a probability $1 - phit_1^{L2} = pmiss_1^{L2}$, called PS_2. In the case i_1 in L2 (PS_1), it does not affect the timing behaviour of i_2. In the case of a miss of i_1 (PS_2) a random number $rMID_1$is stored in $rMID$, which can delay the i_2.

If starting from PS_1, i_2 executes it can proceed normally (with no *EFL* -incured delay) and it creates two new states – PS_3 and PS_4, with an associated probability of $P^{L2}_{hit1} \times P^{L2}_{hit2}$ and $P^{L2}_{hit1} \times P^{L2}_{miss2}$ respectively. If the probabilistic state was PS_2, and further i_2 misses in $L2$, the *EFL* hardware can delay this request. In this

scenario, i_1 and i_2 are consecutive and they are serialised in their access to L2, so the execution time difference (dl_2) in our architecture since the time the first access to the LLC until the second accesses is fixed (5 cycles). Hence, $i2$ request to the LLC will be delayed by *EFL* if the value in $rMID_1$ was bigger than 5.

Same happens with the execution of i_3, *EFL* can delay execution if processor was in probabilistic states PS_4 or PS_6. Overall, after the execution of every instruction we have a state, with an associated probability, in which processor can be. Under this state, the time difference between two LLC misses is fixed. If the random value stored in $rMID$ is bigger than this fixed latency ($rMID_i > dl_i$), the current instruction is delayed, which happens with a given probability ($P_{rMID>dl}$).

Hence the delay introduced by EFL in each probabilistic state of the processor is:

$$lat_{EFL} = \begin{cases} rMID_{i-1} - pd_i & \text{if } (rMID_{i-1} > dl_i) \\ 0 & \text{otherwise} \end{cases}$$

For the case of PS_4, let's call $P_{base} = P^{L2}_{hit1} \times P^{L2}_{miss2}$. In the case of L2 miss, the ETP describing the latency *EFL* caused in i_3 is given by:

$$\begin{aligned} ETP_{EFL} = \{ \quad & \{0, \\ & 1, \\ & 2, \\ & 3, \\ & ..., \\ & 2 * MID - dl_3, \\ & \{P_{base} \times (P_{EFL} \times dl_3), \\ & P_{base} \times P_{EFL}, \\ & P_{base} \times P_{EFL}, \\ & ..., \\ & P_{base} \times P_{EFL}\} \\ \} & \end{aligned} \tag{7.3}$$

Overall, the probabilistic nature of the two sources of delay introduced by *EFL* enables deriving an ETP for all operation in an MBPTA-compliant architecture that deploys *EFL*, which is enough to apply MBPTA trustworthily.

We contrast empirical results by using proper statistical i.i.d tests, as explained in Chapter 3.

Table 7.1 shows the results of the WW and KS tests for all EEMBC benchmarks we used when deploying $EFL250$. For the WW test all results are below 1.96 and for the KS are above 0.05. Hence, for this level of significance ($\alpha = 0.05$), the i.i.d hypothesis can be accepted. WW and KS tests where also passed for all the other

Table 7.1: Results of the i.i.d. tests for the EFL250

Bench.	Identical distribution	Indepen-dence	Both tests passed?
II	0.53	0.44	yes
ID	0.49	0.43	yes
MA	0.18	0.47	yes
PN	0.55	0.76	yes
CN	0.34	0.51	yes
A2	0.59	0.76	yes
AI	0.62	0.59	yes
CA	0.24	0.67	yes
PU	0.25	0.49	yes
RS	0.43	0.73	yes

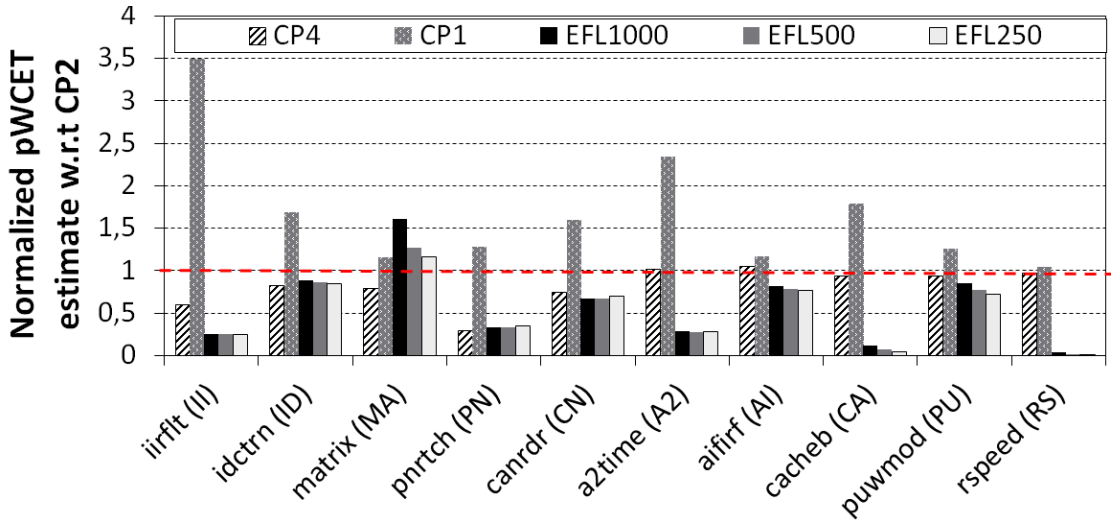


Figure 7.5: pWCET of each setup normalised to CP2

EFL configurations.

***EFL* vs. cache partitioning.** We compare our technique, *EFL*, w.r.t. hardware cache (way) partitioning [Paolieri *et al.* (2009a)]. For that purpose we consider different configurations of both techniques. For *EFL* we consider MID values 250, 500 and 1,000 cycles. We refer to those configurations as EFL*mid*. For cache partitioning we study cache setups with 1, 2 and 4 ways per program. We refer to those configurations as CP*ways*.

Figure 7.5 shows the pWCET estimates for all benchmarks normalised w.r.t. cache partitioning with 2 ways per program, which would be the solution where each of the 4 cores has exactly 2 out of the 8 LLC cache ways.

Some benchmarks (`ID`, `MA`, `CN`, `AI`, `CA`, `PU`, `RS`) are relatively insensitive to

cache space as long as they are given at least 2 ways, i.e. $\frac{1}{4}$ of the cache space. Still *EFL* outperforms *CP* for those benchmarks, since *EFL* does not impose any constraint on the associativity available in each cache set as *CP* does. In particular *EFL* moderately improves *CP* for (`ID`, `CN`, `AI`, `PU`). It is also the case that all those benchmarks but `MA` show to be highly insensitive to the actual *EFL* . `MA` is a benchmark most of whose input set does not fit in LLC. As a result, it experiences a large number of LLC misses and hence, most LLC accesses get delayed due to *EFL*, hence increasing pWCET estimates. In this case it is clear that low MID values mitigate this effect.

Finally, `II`, `PN` and `A2`, which are more sensitive to cache space, are less affected by *EFL* than by *CP*.

Overall, *EFL* clearly outperforms *CP* in terms of pWCET estimates across benchmarks, especially for low MID values.

Guaranteed performance. The principle of both, *CP* and *EFL*, in order to provide guarantees in the performance of tasks is to reserve resources in the LLC, though *CP* does this in a static manner and *EFL* in a probabilistic manner. For a given benchmark, b, by dividing the instructions committed by b and its pWCET estimate, measured in processor cycles, obtained for *EFL* (or *CP*) for a given cutoff probability (e.g. 10^{-15} per run) we obtain b's guaranteed instructions per cycle (gIPC) for that cutoff probability $gIPC^{-15}_{EFL}(b) = \frac{Instructions(b)}{pWCET^{-15}_{EFL}(b)}$. The workload total guaranteed performance, $wgIPC^{prob}$, is obtained by adding the $gIPC^{prob}$ of the benchmarks in the workload.

wgIPC results can be indirectly obtained from Figure 7.5. For instance `A2` with CP4 has a normalized execution time close to 1, while for EFL250 it is 0.27. This translates into the fact that with *EFL* the gIPC of `A2` is almost 4 times bigger than for *CP* . In order to measure the *wgIPC* of *CP* and *EFL* in a systematic way, we randomly generated 1,024 workloads and measured the highest $wgIPC^{-15}$ that *CP* and *EFL* can provide under any setup. For *CP* this is equivalent to find the partition of the 8 ways of the LLC across the tasks such that $wgIPC^{-15}$ is maximised. For the case of *EFL* we look for the MID value (identical for all tasks) that maximises *wgIPC*.

Figure 7.6 shows the improvement that *EFL* obtains over *CP* in terms of *wgIPC*. Workloads are sorted from higher to lower EFL improvement. As exceedance probability we have chosen 10^{-15}, with similar results obtained for 10^{-17} and 10^{-19} .

EFL follows an S-curve and improves *CP* in 1,015 out of the 1,024 workloads. For more than 25% of the workloads improvements are higher than 70%, while for more than half it is higher than 47%. The average degradation for the 10 workloads in which *EFL* is worse than CP is smaller than 3.1% (with a maximum degradation smaller than 9.8%). Overall, *EFL* consistently improves *CP*, with a

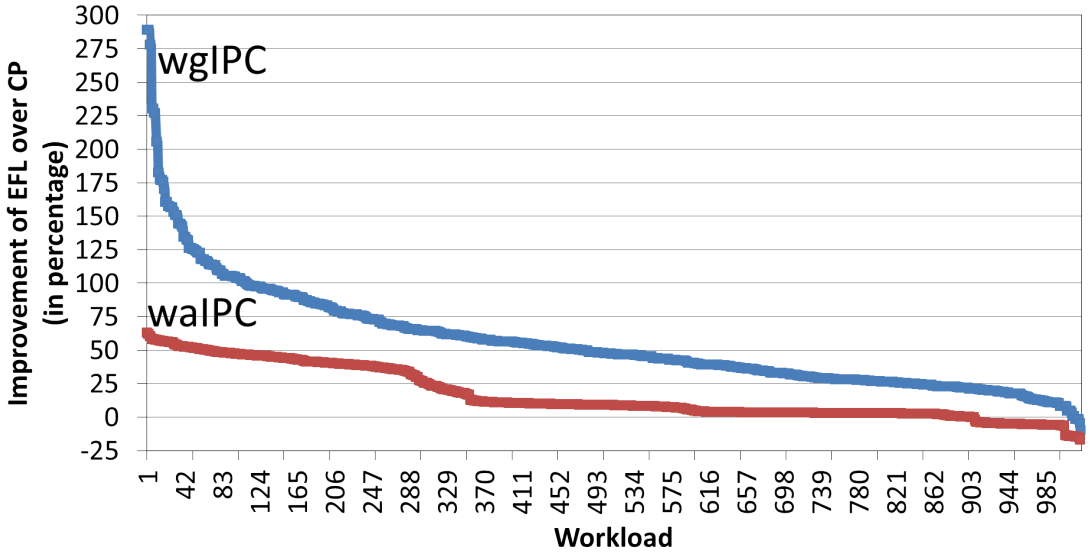


Figure 7.6: Workload guaranteed IPC (*wgIPC*) and average IPC (*waIPC*) improvement of *EFL* over *CP*

maximum *wgIPC* improvement of up to 2.89x and an average of 56%.

Average performance. Average performance is also an important metric in real-time systems. High average performance when running critical tasks enables, for instance, running non-critical tasks or saving power by power-gating cores or decreasing their operating frequency. For the same workloads used in Figure 7.6 we compute their average IPC (*waIPC*) observed at run time. The bottom function in Figure 7.6 shows *EFL* improvement over *CP* in terms of average performance. We observe that *EFL* improves *CP* in 910 out of the 1,024 workloads. For more than 25% of the workloads improvements are higher than 37%, while for more than half of the workloads is higher than 9%. The average degradation for the 10 workloads in which *EFL* is worse than CP is smaller than 6.4% (with a maximum degradation smaller than 16.8%). As for *wgIPC*, *EFL* consistently improves *CP*, with a maximum improvement of up to 64% and an average of 16%.

7.6 Conclusions

Conventional timing analysis techniques rely on cache partitioning, either hardware or software, to obtain time-composable WCET estimates in systems equipped with shared LLCs as they avoid inter-task interferences. However, cache partitioning challenges data sharing and task scheduling.

In this Chapter we propose *EFL*, a new technique to obtain time-composable

WCET estimates on top of shared non-partitioned LLCs, thus removing partitioning constraints. EFL relies on the fact that time-randomised (Evict-on-Miss) caches used in conjunction with PTA remove the dependence of execution time and WCET on the actual addresses accessed by the program. Thus, by controlling when each core is allowed to evict lines from the LLC and by appropriately producing LLC evictions at analysis time, EFL effectively obtains trustworthy and tight WCET estimates. EFL increases the guaranteed performance w.r.t. cache partitioning by 56% and average performance by 16%.

Chapter 8

Reliability issues

Existing timing analysis techniques to derive Worst-Case Execution Time (WCET) estimates assume that hardware in the target platform (e.g., the CPU) is fault-free. The use of smaller transistors helps providing more performance while maintaining low energy budgets; however, hardware fault rates increase noticeably, affecting the temporal behaviour of the system in general, and WCET in particular.

In this Chapter, we propose the *Degraded Test Mode (DTM)*, a method that, in combination with fault-tolerant hardware designs and probabilistic timing analysis techniques, enables the use of hardware implemented with smaller transistors and still provides tight (and trustworthy) WCET estimates.

8.1 Introduction

To respond to the demand for increased computational power in future CRTES, the use of hardware acceleration features such as caches and smaller technology nodes (i.e. smaller transistors) is required. In particular, the higher degree of integration provided by newer technology nodes enables (i) reducing energy consumption and temperature, and (ii) integrating more hardware functionalities per chip, which in turn enables running more system functionalities per chip, thus reducing overall system size, weight and power consumption costs.

Semiconductor technology evolution (i.e. smaller technology nodes) leads, however, to an increased number of permanent faults manifesting during operation [Abella *et al.* (2011a)]. While test methods can detect most permanent faults during post-silicon testing, many rather small defects not producing actual faults escape the test process (latent faults). Hardware degradation makes those latent defects grow enough to cause faults during operation. Errors induced by permanent faults can be tolerated to certain extent in some markets, but cannot in CRTES where stringent correctness constraints call for means to prevent

faults from jeopardising the safety of those systems. While this was not an issue for old technology nodes (e.g., 180nm), it becomes critical for newer nodes (e.g., 65nm) since Failures in Time (FIT)[1] rates may grow quickly after only 10 years of operation [Guertin & White (2010)], which is less than the lifetime required for many CRTES (e.g., aircraft, space missions). Furthermore, smaller technology nodes (45nm and beyond) suffer high FIT rates much earlier. Some existing error detection and correction techniques can cope with both functional and *timing* correctness required in CRTES despite of permanent faults. For instance, triple modular redundancy (TMR) [Lyons & Vanderkulk (1962)] has no effect on the timing behaviour despite of errors, but introduces high overheads since complete hardware blocks (e.g., processing cores) are replicated. Analogously, other approaches based on setting up some degree of redundancy are expensive in the context of CRTES if used extensively to deal with permanent faults [Chen & Hsiao (1984),Koren & Koren (1998)]. Alternatively, hardware can be reconfigured (e.g., disabling faulty components) when permanent faults arise [Abella *et al.* (2009),Roberts *et al.* (2007),Wilkerson *et al.* (2008)]. However, such reconfiguration provides degraded performance, which is a major concern in CRTES, since CRTES must provide both, functional and *timing* correctness.

The fact that hardware characteristics degrade in an exponential number of different ways due to faults challenges state-of-the-art timing analysis methods. Static Timing Analysis (STA) does not take into account any information about the permanent faults that hardware may experience. So, as soon as any of the hardware components (e.g., a cache line) is detected to be faulty and disabled by the fault-tolerance mechanisms in place, the WCET estimates previously derived are no longer a safe upper-bound of the WCET of the application. Similarly, WCET estimates derived with Measurement-Based Timing Analysis (MBTA) techniques or Hybrid ones (i.e. combination of MBTA and STA) [Wilhelm *et al.* (2008)], remain only valid under the same processor degraded mode on which measurements were taken (typically a fault-free mode).

In summary, it becomes mandatory for timing analysis tools to account for degraded hardware behaviour. However, to the best of our knowledge no timing analysis technique has been developed so that it can safely and tightly provide WCET estimates on top of degraded hardware. Only few works have proposed ad-hoc hardware solutions to keep timing characteristics of hardware despite faults at the expense of some redundancy and extra complexity in cache memories, which experience most hardware faults due to their large area and aggressive transistor scaling to provide further cache space and reduced energy consumption [Abella *et al.* (2011b),Abella *et al.* (2011c)]. Also, some work has been done to account for degraded hardware together with STA [Hardy *et al.* (2016)]. However, such

[1] 1 FIT corresponds to 1 failure per 10^9 hours of operation.

work needs to account for either fault maps whose number grows exponentially with fault counts, or for very pessimistic assumptions (e.g. a fault affects all cache sets) to limit computational cost.

This Chapter addresses the challenges of enabling tight and safe WCET estimation on faulty hardware with graceful WCET degradation. The contributions of this Chapter can be summarized as follows:

1. We propose the *Degraded Test Mode* (DTM) approach to provide strong timing guarantees on systems with degradated cache memories due to permanent faults.

2. We specify how hardware must be designed and tested so that faults are tolerated enabling the use of MBPTA on top of such hardware.

3. We propose a holistic approach to deal with the timing impact of error detection, correction, diagnosis and reconfiguration (DCDR) on systems with cache hierarchies.

DTM focuses on cache designs, since most faults are expected to affect caches and caches are one of the resources affecting WCET tightness the most. DTM determines the properties cache memories must exhibit and how they must be configured so that they can be analysed with PTA techniques, while obtaining WCET estimates valid in the presence of faults. DTM achieves (i) graceful average performance degradation and (ii) WCET degradation in the presence of faults, while still enabling (iii) trustworthy and tight WCET estimates despite of faults, by (iv) using unmodified MBPTA tools and (v) introducing no changes in applications so that DTM can be used even for legacy code.

When considering a holistic approach for DCDR, we also account for faults in other processor components apart from cache memories.

8.2 Background

CMOS technology suffers from process variations [Bowman *et al.* (2002)]. Process variations are deviations of device (e.g., transistor, wire) parameters from their nominal values. The relative impact of those variations increases as device geometry shrinks. While this is a challenge for all hardware components, it is particularly critical for cache memories because bit-cells are typically implemented with the smallest transistors allowed (so the relative impact of variations exacerbates). Moreover, variations cannot compensate across devices because bit-cells involve very few transistors (e.g., typical cells are implemented with 6 or 8 transistors only [Jain & Agarwal (2006)]). Deviations due to process variations lead to

timing faults (signal transitions take longer than the cycle time), retention faults (bit-cell contents eventually flip from 0 to 1 or vice versa) and read/write faults (bit-cells cannot be properly read or written).

Test methods exist to verify that processors are fault-free when delivered. Unfortunately, latent defects are not large enough to cause errors and escape this test process. After some time of operation, degradation makes latent defects to cause actual faults. This effect exacerbates as device geometry shrinks. To illustrate this phenomenon Figure 8.1 shows the FIT rate for technologies down to 65nm [Guertin & White (2010)]. We can observe that until 10^4 hours of operation the FIT rate decreases. This period is known as *infant mortality* and is elapsed before product deployment through accelerated stress in appropriate equipment. Then, the FIT rate remains low during some time (normal lifetime) until it raises exponentially due to degradation. Rightmost vertical lines in the plot show how this FIT rate ramp up occurs earlier for newer technologies, leading to a too short normal lifetime for some technologies (e.g., few thousands of hours for 65nm). Therefore, the normal lifetime where FIT rates are low enough for safety-critical systems become too short. For instance, while such ramp up would take hundreds of years to occur for 180nm technology nodes, it occurs after 10 years of operation for 65nm. This is already a challenge for avionics and space industry where aircraft and space missions are expected to last several decades (i.e. $8 \cdot 10^4 - 4 \cdot 10^5$ hours), and where the FIT rate is expected to be largely below 100), as shown in the dotted green rectangle in Figure 8.1. Further, 45nm nodes and beyond are expected to observe a FIT rate ramp up much before 10 years of operation ($8 \cdot 10^4$ hours), thus challenging also other CRTES industries such as automotive and medical ones.

Several techniques exist to perform error detection, correction, diagnosis and reconfiguration (DCDR), so functional correctness, as required in CRTES, can be achieved. However, modifying hardware behaviour to keep operating in the face of faults (e.g., disabling faulty cache lines or decreasing operating frequency) degrades performance. Unfortunately, existing timing analysis methods are unable to provide tight and trustworthy WCET estimates if hardware is expected to become faulty during operation. The fact that tasks are executed correctly but late (finishing after their respective deadlines) can be as catastrophic in CRTES as producing wrong results. Therefore, methods to account for permanent faults in WCET estimates are needed.

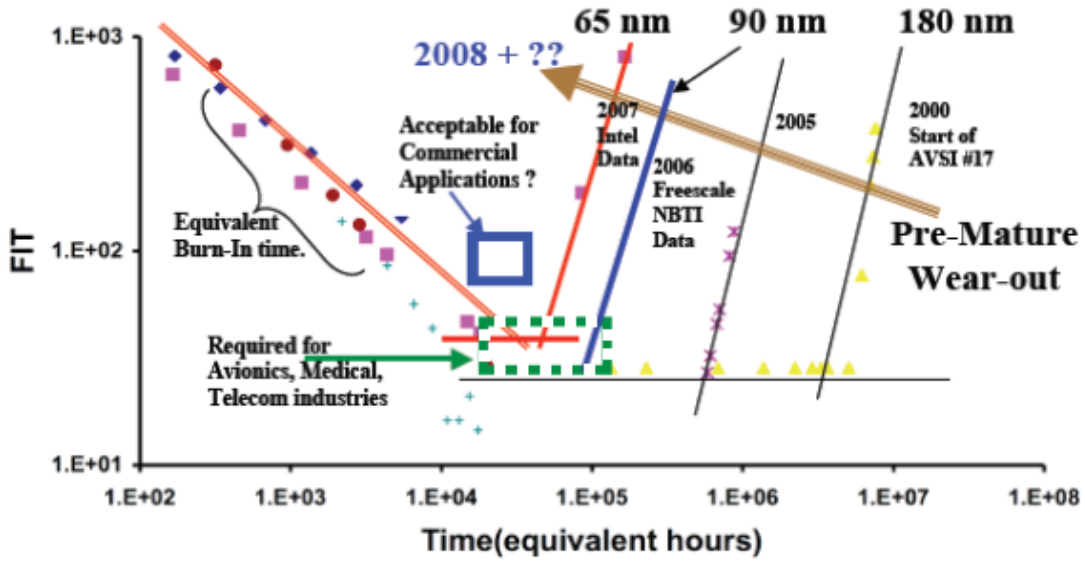


Figure 8.1: FIT rates of modern semiconductor technologies used for embedded microprocessors (65nm, 90nm, 130nm and 180nm) over the chip's lifetime. The X and Y axes show equivalent hours of operation and FIT rates respectively in logarithmic scale (source: Jet Propulsion Laboratory, NASA [Guertin & White (2010)]).

8.3 Target Processor Architecture and Fault Model

This section introduces the hardware designs upon which we build our approach as well as the fault model used.

8.3.1 Hardware Designs for MBPTA and Caches

Our processor architecture consists of a 4-stage (fetch, decode, execute and write-back/commit) in-order processor with first level data (DL1) and instruction (IL1) caches, as well as data (DTLB) and instruction (ITLB) translation lookaside buffers. DL1, IL1, DTLB and ITLB can be fully-associative, set-associative or direct-mapped as long as they implement random placement and replacement [Kosmidis *et al.* (2013a)]. The only constraint DTM puts is that the processor must be MBPTA-friendly so that MBPTA can be used to determine the pWCET.

Although any component in the processor can be faulty, we only consider faults in cache-like components such as DL1, IL1, DTLB and ITLB since they are expected to concentrate most of the faults as explained in Section 8.2. However, the

approach proposed in this Chapter can consider faults in any component as long as those faults do not impact the functional correctness of programs being executed —faults only impact timing once they have been detected and faulty parts reconfigured properly.

If cache-like blocks are expected to experience soft errors, they may be parity or Error Detection and Correction (EDC) protected. We consider that those error detection and/or correction features are set up *only* to provide a given degree of fault tolerance in front of soft errors, whose rate can be very high in segments such as avionics and space, where systems operate in high-radiation environments (thousands of meters above the sea level or the space) and how our proposal can deal with them is explained later in Section 8.6.2. Permanent faults arising from the use of nanotechnologies cannot be addressed by those means set up to deal with soft errors because that would decrease soft error protection. For instance, if data in cache are Single-error-correction and double-error-detection (SECDED) protected to be able to correct up to one soft error per word, permanent faults can rely on SECDED to detect and correct the error the first time it manifests, but those words must be disabled because a soft error combined with the existing permanent fault could easily lead to a non-correctable double error.

We assume that whenever a faulty bit is detected in cache, the whole cache line is disabled. Such an approach is similar to what we can find in some existing (non-embedded) processors such as the Itanium family, which implements Pellston technology [McNairy & Mayfield (2005)] to disable faulty cache lines during operation. A potential implementation of Pellston technology consists of extending each cache line with a *Usable Bit* (UB for short). Such bit must be set for all cache lines but faulty ones. Whenever cache tags are checked on an access, the match signal is ANDed with the UB so that faulty lines cannot report a hit. Note that such AND operation is also performed with the valid bit, which indicates whether the cache line holds valid contents.

8.3.2 Permanent Fault Model

In order to model permanent faults caused by process variations and aging in nanotechnologies, we use as input the fault probability per bit ($p(bit)_f$), which is the common practice in fault modeling in caches due to the random nature of those faults [Abella *et al.* (2009),Roberts *et al.* (2007),Wilkerson *et al.* (2008)]. Such $p(bit)_f$ corresponds to the probability of a bit to become permanently faulty during the target lifetime of the chip. We assume faults to be random given that systematic (and hence, non-random) faults can be easily addressed with techniques such as body biasing [Tschanz *et al.* (2002)]. Hence, we apply such $p(bit)_f$ homogeneously to all DL1, IL1, DTLB and ITLB bits except the UB bits, which must be hardened to allow faulty lines to be properly disabled. Hardening can

be performed, for instance, by using larger transistors or triple modular redundancy. Based on the $p(bit)_f$ and the number of bits per line ($bits_{line}$) we obtain the probability of a line to be fault-free ($p(line)_{ok}$) in each cache-like structure:

$$p(line)_{ok} = (1 - p(bit)_f)^{bits_{line}} \tag{8.1}$$

Next, we obtain for each cache-like structure the probability of each faulty cache line count: $p(cache)_x$ where x is the number of faulty cache lines and N the total number of cache lines:

$$p(cache)_x = (p(line)_{ok})^{N-x} \cdot (1 - p(line)_{ok})^x \cdot \begin{pmatrix} N \\ x \end{pmatrix} \tag{8.2}$$

The first element in Equation 8.2 corresponds to the probability of $N - x$ lines not to be faulty, the second element is the probability of having x lines faulty and the third the number of arrangements of x faulty lines in a cache with N lines.

8.3.3 Upper-bounding the Number of Faulty Cache Lines

Based on the probabilities of each faulty cache line count, we derive an upper-bound for the number of cache lines that must be assumed to be faulty during the timing analysis to derive safe WCET estimates. For that purpose, we derive the probability of a cache to have up to x faulty lines. We refer to this probability as $Yield(cache)$:

$$Yield(cache) \leq \sum_{i=0}^{x} p(cache)_i \tag{8.3}$$

where x is the faulty cache line count. Note that Equation 8.2 gives the probability of an exact number of cache lines (x) to be faulty. Equation 8.3 accumulates the probability for those counts up to x faulty lines. For instance if $x = 3$, $Yield(cache)$ stands for the probability of having at most 3 faulty lines in cache during the whole lifetime of the chip.

Finally, the yield of the chip, $Yield(chip)$, can be computed based on the yield of its components (cache memories in our case study).

$$Yield(chip) = \prod_{i=1}^{NumCaches} Yield(cache_i) \tag{8.4}$$

$NumCaches$ stands for the total number of cache memories in the chip. Such yield must be high enough so that the failure rate (obtained as $1 - Yield(chip)$)

Inputs:
Outputs:
(1) C = set of all caches
(2) X = set of faulty entries considered for each cache
(3) for each cache $cache_i \in C$ do
(4) $x_i = 0$ (where $x_i \in X$)
(5) while $(1 - Yield(cache_i)) > TargetFailureRate(chip)$ do
(6) $x_i = x_i + 1$
(7) endwhile
(8) endfor
(9) while $(1 - \prod_{i=1}^{NumCaches} Yield(cache_i)) > TargetFailureRate(chip)$ do
(10) $cache_{max} \in C$ so that $\forall cache_i \in C : Yield(cache_{max}) \leq Yield(cache_i)$
(11) $x_{max} = x_{max} + 1$
(12) endwhile
(13) return X

Figure 8.2: Algorithm to obtain the number of faulty entries to consider for each cache.

is below a given target threshold ($TargetFailureRate(chip)$) as shown in Equation 8.5.

$$TargetFailureRate(chip) \geq 1 - Yield(chip) \tag{8.5}$$

Note that $TargetFailureRate(chip)$ is very stringent for CRTES (e.g., 10^{-6}, meaning that up to 1 chip every 1,000,000 may have more faults than allowed during its lifetime).

When the chip comprises several cache memories, several combinations of faulty entries per cache may meet the $TargetFailureRate(chip)$. For instance in a processor with DL1, IL1, DTLB and ITLB the maximum number faulty lines that must be considered for each of those cache memories could be $\{3, 3, 2, 1\}$ or $\{4, 3, 1, 1\}$.

The number of faulty cache lines we must assume for each cache component must be determined based on the $TargetFailureRate(chip)$. How to do this simultaneously for all caches is not trivial, so we develop an algorithm to derive the number of faulty lines per cache that must be considered for WCET estimation. We start considering the maximum number of faulty lines per cache memory such that each particular cache in isolation does not exceed the target failure rate for the whole chip. If the combined yield of the different caches exceeds the target failure rate, then we need a way to increase the number of faulty lines considered

in the different caches so that $Yield(chip)$ is increased. For that purpose we propose a simple iterative algorithm to deal with those scenarios. The algorithm is presented in Figure 8.2. First, the number of faulty cache lines to be considered for each cache in isolation is computed (lines 3-8). Then, the yield of each cache in isolation is combined. If the combined failure rate $(1 - Yield(chip))$ exceeds the threshold (line 9), an extra faulty entry is assumed for the cache with the lowest yield (lines 10-11). This process repeats until their combined failure rate is below the target threshold (lines 9-12).

8.4 Making Timing Analysis Aware of Hardware Faults

8.4.1 Deterministic Hardware

STA techniques require detailed knowledge of the underlying hardware. For instance, in the case of cache analysis, STA must be aware of the replacement policy (e.g., LRU) and the placement policy (e.g., modulo) used by the hardware. From the application executable, STA needs the address of each cache access to be able to determine whether they will hit or miss in cache.

Let us assume that, based on the model presented in Section 8.3.2, we know that the cache deployed in a given platform is expected to have up to f faults during its lifetime. In order to make STA tools aware of faults at hardware level we should consider all combinations of f faults over N total lines in cache, leading to a total of $\frac{N!}{f!\cdot(N-f)!}$ different degraded cache modes[1]. For instance, if up to 2 faults are expected ($f = 2$) in a small cache with 64 cache lines ($N = 64$), STA should consider 2,016 different scenarios. Under each degraded mode, cache analysis should be carried out deriving a WCET bound. The highest of those WCET bounds should be taken as the final WCET bound. MBTA techniques would suffer similar problems, since the particular lines in which faults appear affect the analysis.

Overall, conventional caches deploying deterministic placement and replacement policies such as modulo and LRU make WCET estimates provided by STA and MBTA depend on the particular location in which faults occur. In the presence of f faults in a cache with N total cache lines, all possible $\frac{N!}{f!\cdot(N-f)!}$ degraded modes must be considered by the timing analysis technique.

[1]The particular location of the faulty line matters given that its particular cache set will have one line less than the other sets, and the replacement policy will select cache lines for replacement regardless of their faultiness, thus leading to different scenarios depending on the particular location of the faulty cache line in the set.

8.4.2 Probabilistic (Time-Randomised) Hardware

Randomised caches (i.e. caches deploying random placement and random replacement) [Kosmidis *et al.* (2013a)] break the dependence between the location in memory of a piece of data and its assigned set in cache.

The main property of MBPTA-friendly (a.k.a. randomised) cache designs is that they are insensitive to the particular location of faults. This drastically simplifies capturing the effect of faulty lines with MBPTA techniques. In particular, the only information to take into account by MBPTA techniques to handle faulty cache lines is *the number of faulty lines in each cache-like structure, not their location in cache (i.e. the particular way and set in which the fault line appears)*. We illustrate this phenomenon in the following subsections, starting with a fully-associative random-replacement cache and then extending it to more generic set-associative caches.

Fully-associative caches

Random replacement is performed by generating, on a miss, a random number of $\log_2 N$ bits where N is the number of cache lines. On an eviction, the random number obtained can correspond to the cache line number of a faulty cache line. If this is the case, a new random number is generated until the identifier of a fault-free cache line is generated. Note that the expected number of attempts (ATT) required for a cache with N total cache lines and f faulty cache lines (where $f < N$) to pick a fault-free cache line is as follows [Feller (1966)]:

$$ATT = \sum_{i=0}^{\infty} \left(\frac{N-f}{N}\right) \cdot \left(\frac{f}{N}\right)^{i} \cdot (i+1) = \frac{N}{N-f} \tag{8.6}$$

In the equation, $\left(\frac{N-f}{N}\right)$ is the probability of picking a fault-free line, and $\left(\frac{f}{N}\right)^{i}$ the probability of picking a faulty line. Hence, the equation adds each number of attempts $(i+1)$ weighted by the probability of i failed attempts followed by 1 successful attempt. This can be expressed simply as $\frac{N}{N-f}$.

For instance, in a cache with $N = 64$ cache lines and $f = 2$ faulty lines we would need around 1.03 attempts on average. Those extra attempts are unlikely to impact the execution time because they can occur in parallel with the memory access. In our example where 2 out of 64 cache lines are faulty, 1 extra attempt is needed with 0.03 probability, 2 extra attempts with 0.001, 3 extra attempts with 0.00003, and i extra attempts with $\left(\frac{N-f}{N}\right) \cdot \left(\frac{f}{N}\right)^{i}$ probability.

Note that those cache designs considered for DTM (caches implementing random placement and replacement with re-try support for evictions of faulty lines), differently to existing approaches [Abella *et al.* (2011b),Abella *et al.* (2011c)], neither need any cache line redundancy nor additional structures. Moreover, as shown

later, the proposed cache designs can be considered in the context of MBPTA because whether a faulty cache line is chosen for replacement depends solely on a random event.

Set-associative caches

If the cache under consideration is set-associative, Equation 8.6 applies at the level of cache set, with N being the associativity of the cache (a.k.a. the number of cache lines per set). A further consideration must be done for set-associative caches, especially if their degree of associativity is low. If the number of potential faulty cache lines is equal or larger than the associativity of such a cache, then exists a non-null probability of having a cache set without any fault-free cache line. If this is the case, accesses to that cache set will be processed as misses and, obviously, not cached. However, given that the number of faulty lines to consider is pretty low, as shown later, and the probability of concentrating many faulty cache lines in a single set is also very low, this scenario is extremely unlikely — if at all possible.

As stated before, randomised caches make MBPTA and DTM insensitive to the location of faults in a particular cache-like structure. Instead, *what really matters for MBPTA and DTM is the number of faulty lines in each cache-like structure, not their location*, so the fault model in Section 8.3 captures exactly the relevant information for MBPTA and DTM.

8.4.3 DTM: Applying MBPTA on Top of Faulty Hardware

Once determined the number of faulty entries that must be considered in each cache structure, see Section 8.3.3, MBPTA must be properly used to determine the pWCET of the program in such a potentially faulty chip.

We collect execution times of the application as dictated by MBPTA [Cucu-Grosjean *et al.* (2012)], but on top of the hardware with maximum degradation. In other words, execution times must be collected on top of a processor with as many cache entries disabled in each cache structure as determined by the process in Section 8.3.3. For that purpose we propose enabling a *Degraded Test Mode* (*DTM* for short) that allows disabling a number of cache entries in each cache structure.

The *DTM* affects some parameters of the processor operation similar to those set up in the BIOS or to those that can be configured dynamically in many processors such as the operating frequency. Therefore, *DTM* can be configured through the BIOS at boot time or through special purpose registers modified during operation. If the latter approach is used, whenever a cache line is deactivated its

contents must be written back to the corresponding level of the memory hierarchy if they are dirty.

Given that PTA (and hence MBPTA) is insensitive to the location of faults, simply configuring the number of disabled entries per structure suffices and any implementation of such disabling process will produce equivalent results from a PTA perspective. For instance, cache entries disabled in a fully-associative cache can be either consecutive or randomly located. In a set-associative, whether several faults occur in the same cache set may impact execution time. Thus, it is still irrelevant in which particular cache set faults occur, but not whether they occur in the same or different cache sets. In particular, in this work we assume that disabled cache lines are randomly chosen. The particular lines disabled are changed randomly before each execution of the program. This change can be performed with a specific instruction set up in purpose.

Once the appropriate number of entries is disabled in each cache structure, MBPTA can be used as in fault-free processors running the program under analysis as needed. *By disabling a number of entries in each of the cache memories, DTM allows collecting execution times in a fault-free processor as if it was faulty so that pWCET estimates are trustworthy regardless of whether hardware operation has been degraded due to faults.*

Although DTM is only described for faulty caches (our case study in this Chapter), the same rationale can be used to deal with faults in other components. Some discussion on this matter is provided later in Section 8.6.5.

8.5 Evaluation

This section describes the evaluation framework and provides results proving the suitability of the proposed platform to be used in the context of faulty processors.

8.5.1 Evaluation Framework

Like in the previous proposals in this Thesis, execution times have been collected with the SoCLib simulation framework [Pouillon *et al.* (2009)] and using EEMBC Autobench benchmark suite [Poovey *et al.* (2009)]. Two different configurations have been considered:

1. Both DL1 and IL1 caches are 2KB fully-associative random replacement 32B/line caches. Both the DTLB and ITLB are 16-entry fully-associative random replacement TLBs for page sizes of 1KB.

2. Both DL1 and IL1 caches are 2KB 4-way set-associative random placement and replacement 32B/line caches. Both the DTLB and ITLB are 16-entry 4-

Table 8.1: Maximum number of faulty lines expected for a target yield of 1 faulty part per million (ppm)

$p(bit)_f$	DL1	IL1	DTLB	ITLB
10^{-8}	1	1	1	1
10^{-7}	2	2	1	1
10^{-6}	3	3	1	1
10^{-5}	4	4	2	2
10^{-4}	10	10	3	3

way set-associative random placement and replacement TLBs for page sizes of 1KB.

Cache sizes are deliberately small so that disabling some cache lines due to faults has some significant effect in performance given that benchmarks used have particularly small working sets. Hit latencies of 2 cycles and memory latencies of 100 cycles have been considered for all caches.

8.5.2 Worst-Case Execution Time

Based on the fault model described in Section 8.3.2 we have obtained how many DL1, IL1, DTLB and ITLB entries can be faulty to guarantee that at most one out of one million chips has more failures than budgeted during its lifetime or, in other words, the failure rate during the lifetime of the chip is below 10^{-6}. Results are shown in Table 8.1.

Tables 8.2 and 8.3 report pWCET relative results for fully-associative and set-associative caches respectively. The minimum number of runs per benchmark has been computed with the method reported in [Cucu-Grosjean *et al.* (2012)] and never exceeded 1,000 runs. *Note that this is the same number of runs needed for a fault-free system, thus keeping pWCET estimation cost low.* Similarly, independence and identical distribution tests described in [Cucu-Grosjean *et al.* (2012)] have been run and all data passed those tests successfully, as needed by MBPTA.

The 2^{nd} column in tables 8.2 and 8.3 shows the pWCET of a fault-free chip for an exceedance threshold of 10^{-15} per run normalised to the average execution time for a fault-free chip[1]. The exceedance threshold indicates how often a run of

[1]Note that the exceedance probability (10^{-15} per run in our case) and the faulty bit rate ($p(bit)_f$) stand for different concepts. $p(bit)_f$ determines the number of faulty cache lines that must be considered for each cache memory. Then, those (degraded) cache memories are used to obtain execution times, which are the input of MBPTA. Finally, given the pWCET distribution

Table 8.2: pWCET normalised results for fully-associative caches

	pWCET fault-free	**pWCET** $p(bit)_f$ 10^{-8}	**pWCET** $p(bit)_f$ 10^{-7}	**pWCET** $p(bit)_f$ 10^{-6}	**pWCET** $p(bit)_f$ 10^{-5}	**pWCET** $p(bit)_f$ 10^{-4}
	vs avg	**vs pWCET fault-free**				
a2time	1,086	1,060	1,120	1,185	1,250	1,614
aifirf	1,035	1,012	1,021	1,014	1,031	1,074
cacheb	1,093	1,027	1,002	1,019	1,010	1,039
canrdr	1,035	0,990	0,990	0,995	1,002	1,004
puwmod	1,014	1,002	1,004	0,999	1,000	1,005
rspeed	1,035	1,017	1,005	1,004	0,998	1,009
tblook	1,057	1,048	1,063	1,119	1,151	1,369
ttsprk	1,034	1,015	1,011	1,004	1,003	1,013
AVG	1,049	1,022	1,027	1,042	1,056	1,141

Table 8.3: pWCET normalised results for set-associative caches

	pWCET fault-free	**pWCET** $p(bit)_f$ 10^{-8}	**pWCET** $p(bit)_f$ 10^{-7}	**pWCET** $p(bit)_f$ 10^{-6}	**pWCET** $p(bit)_f$ 10^{-5}	**pWCET** $p(bit)_f$ 10^{-4}
	vs avg	**vs pWCET fault-free**				
a2time	4,709	1,157	1,190	1,301	1,693	2,656
aifirf	4,356	1,046	1,075	1,147	1,265	1,335
cacheb	2,351	1,017	1,051	1,143	1,148	1,179
canrdr	1,160	1,007	1,033	1,060	1,065	1,101
puwmod	1,037	0,990	0,997	1,018	1,028	1,031
rspeed	1,049	1,011	1,059	1,070	1,136	1,143
tblook	2,536	1,105	1,135	1,139	1,328	1,358
ttsprk	1,159	1,029	1,032	1,040	1,047	1,069
AVG	2,295	1,045	1,071	1,115	1,214	1,359

the program will exceed the pWCET value (in our case once every 10^{15} runs).

We can observe in the 2^{nd} column of Table 8.2 that the pWCET with fault-free caches degrades only between 1% and 10% across benchmarks (5% on average) w.r.t. the average performance for fully-associative caches in line with results in [Cucu-Grosjean *et al.* (2012)]. Conversely, the pWCET for a set-associative cache is significantly larger than the average execution time as described in [Kosmidis *et al.* (2013a)] (2^{nd} column of Table 8.3). The reason behind this behaviour

provided by MBPTA, we pick as pWCET estimate the pWCET value corresponding to the particular exceedance probability required.

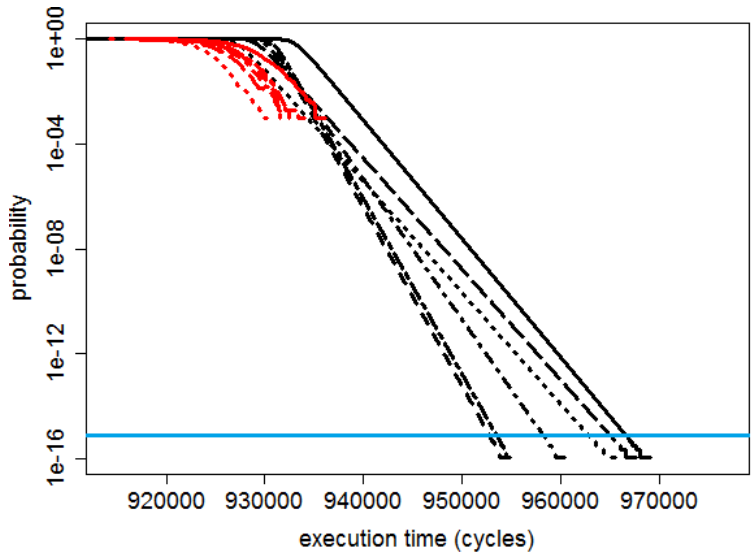


Figure 8.3: Inverse cumulative distribution function (ICDF) of pWCET curves and actual observations for *canrdr* benchmark under a fully-associative random-replacement cache. At the 10^{15} probability cutoff point, from left to right, cache configurations cross in the following order: $p(bit)_f = 10^{-8}$, $p(bit)_f = 10^{-7}$, $p(bit)_f = 10^{-6}$, *fault-free*, $p(bit)_f = 10^{-5}$, $p(bit)_f = 10^{-4}$.

is the fact that fully-associative caches perform a random eviction on every miss. Instead, set-associative caches randomly map each address into a cache set, and such mapping holds during the whole run. Therefore, if some addresses collide in a particular cache set, this behaviour will hold during the whole execution, so its impact in execution time will be larger. Such larger variation requires a higher pWCET upper-bound than for fully-associative caches. This can be easily explained with an example. Let assume we flip 3 coins. If we flip each one individually (random replacement), then we have 8 different sequences of events. Instead, if we attach the 3 coins with glue (e.g., faces in one side and tails in the other side) and then we flip them (random placement), only 2 different events can occur. Thus, although both approaches are equally random, the potential scenarios that they can generate are different (random placement can effectively generate only a subset of those scenarios produced by random replacement) and so their probabilities.

Columns from 3^{rd} to 7^{th} in Tables 8.2 and 8.3 show the pWCET degradation w.r.t. the pWCET of a fault-free chip for different $p(bit)_f$ values. As shown most of the benchmarks observe negligible pWCET degradation because they do not fully exploit all cache space[1] and suffer no degradation due to the deactivation of

[1]Some benchmarks like *aifirf* do not fit in cache, so they reuse few cache lines. In this case,

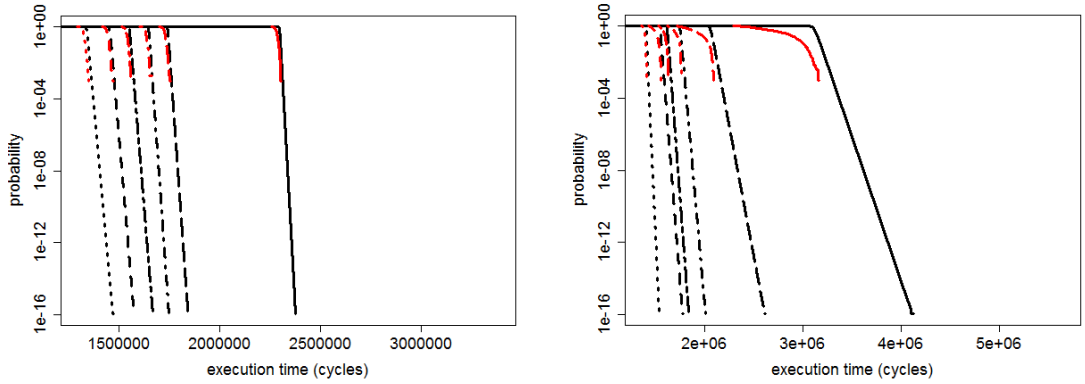
few cache entries. This is particularly true for fully-associative caches (Table 8.2). In fact *canrdr* observes some negligible pWCET reductions in some cases. This occurs because of the particular 1,000 execution times in the sample, but results are still safe and tight. We show the ICDF for the different $p(bit)_f$ pWCET curves of *canrdr* together with actual execution times collected for a fully-associative random-replacement cache in Figure 8.3. At 10^{-15} exceedance probability, the *fault-free* case, which one would expect to provide the lowest pWCET value, provides in fact the fourth lowest value. However, as already discussed in [Cucu-Grosjean *et al.* (2012)], this can happen because MBPTA builds upon a finite random sample. Therefore, although the method provides a Gumbel distribution upper-bounding the tail of the actual execution time distribution, the tightness of the particular distribution may vary depending on the actual observations in the sample. In the particular case of *canrdr*, cache size has negligible impact in performance and, therefore, having fewer cache lines has an effect on performance largely below that of the actual random events occurring during the execution (i.e. random replacement of cache lines). In the particular case of the *fault-free* cache, this leads to a larger σ value for the Gumbel distribution, which increases the overestimation as the exceedance probability decreases. Note, however, that such overestimation is around only 1% higher than the actual overestimation[1] for $p(bit)_f = 10^{-8}$, $p(bit)_f = 10^{-7}$ and $p(bit)_f = 10^{-6}$ cases.

pWCET estimates increase for cache-sensitive benchmarks (*a2time* and *tblook*) in the fully-associative configuration as shown in Table 8.2. A maximum pWCET increase of 61% is needed for *a2time* despite the high $p(bit)_f$ values considered due to the performance robustness of PTA-friendly cache designs. pWCET estimates for set-associative caches (see Table 8.3) grow faster with faulty bit rates due to the increased variability caused by those cache lines being disabled. This effect is particularly noticeable for those benchmarks using efficiently the cache space available such as *a2time*, *aifirf* and *tblook*. However, even for high $p(bit)_f$ values (e.g., $p(bit)_f = 10^{-5}$), pWCET degrades only 21% on average with respect to the fault-free scenario.

For the sake of illustration, we show the different ICDF curves for *a2time* for both cache configurations in Figure 8.4a (fully-associative cache) and Figure 8.4b (set-associative cache). In both cases curves appear from left to right, starting with the lowest $p(bit)_f$ value (0 for the *fault-free* case) and ending with the highest $p(bit)_f$ value ($p(bit)_f = 10^{-4}$). As already shown in [Cucu-Grosjean *et al.* (2012)], pWCET distributions upper-bound actual execution times in the probability range

most of the cache lines become useless, as it is the case of those benchmarks needing less cache space than available.

[1]Note that the actual overestimation cannot be determined because computing the exact execution time distribution is unaffordable in general. In [Cucu-Grosjean *et al.* (2012)] authors prove for a particularly simple scenario that such overestimation is rather low.



(a) Fully-associative placement

(b) Set-associative placement

Figure 8.4: Inverse cumulative distribution function (ICDF) of pWCET curves and actual observations for *a2time* benchmark under different placement functions

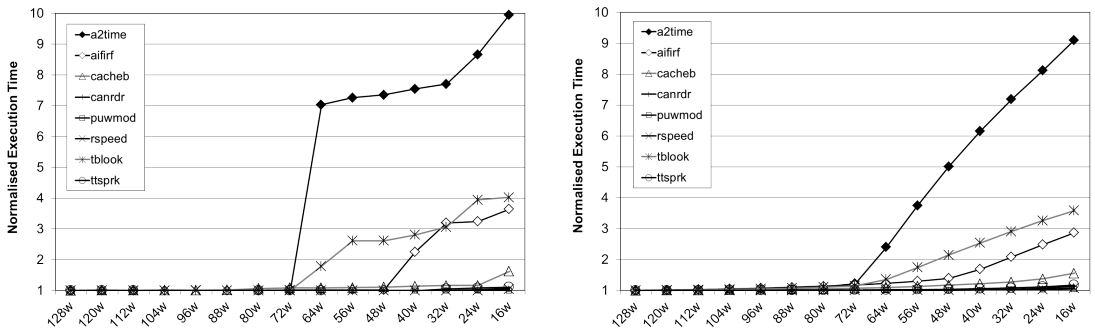
of those execution times (i.e. down to 10^{-3} exceedance probability per run in our case). pWCET curves upper-bound real distributions for any exceedance probability. Their tightness depends on the actual variability existing in the execution times collected. For instance, variability is low in the case of fully-associative caches because each replacement occurs independently as reported in [Kosmidis *et al.* (2013a)]. This leads to very tight pWCET estimates. However, if variability is higher (e.g., the case of a random-placement cache when $p(bit)_f = 10^{-4}$), MBPTA ends up deriving a Gumbel distribution accounting for such variability in the form of a higher σ value, which translates into a gentler slope and so, potentially less tight pWCET estimates.

In summary, MBPTA together with MBPTA-friendly cache designs are particularly suitable to obtain time-robust and time-analysable fault-tolerant hardware designs as needed for DTM. *DTM enables for the first time the computation of WCET estimates on top of faulty hardware at very low cost.*

8.5.3 Average Performance

This section analyses the average performance for fully-associative caches. While this could be done also for set-associative caches, fully-associative ones allow varying cache size at fine granularity (e.g., removing one cache line), whereas set-associative caches could only be studied at a coarser granularity (e.g., removing one cache way or half of the cache sets). Nevertheless, conclusions for fully-associative caches can be easily extrapolated to set-associative ones.

As stated before, caches implementing LRU replacement are not suitable for PTA and, instead, random-replacement (RR for short) ones must be used. In this



(a) LRU-replacement (b) random-replacement

Figure 8.5: Normalised execution time for fully-associative caches with different replacement functions, with respect to a 128-line cache.

section we show that RR caches provide performance comparable to LRU ones and, on top of that, provide much better average performance degradation when cache size is decreased. Such a feature is particularly interesting for WCET analysis on top of faulty caches because disabling few cache lines due to faults has relatively low impact in performance.

Figures 8.5a and 8.5b show the degradation of the execution time of several EEMBC benchmarks when the number of cache lines for DL1 and IL1 caches decreases for LRU and RR caches respectively. Cache size is decreased by 8 lines in each step from 128 (4KB) down to 16 cache lines (512B). Although such granularity is coarser than the number of faulty cache lines one may expect in those caches, it serves to illustrate how LRU caches experience abrupt performance degradation when the working set does not fit in cache, whereas such degradation is much smoother for RR caches. Although not shown, we have also observed that decreasing cache size by 1 line in each step still provides smooth performance degradation for RR caches and abrupt changes for LRU ones. For instance, the epitome example is *a2time*, whose execution time grows by 7X when reducing cache size from 72 to 64 caches lines. RR caches suffer the same relative performance degradation when reducing cache size from 72 to 32 cache lines, thus smoothing the effect of some cache lines being disabled. At a lower scale *aifirf* and *tblook* show similar effects. By smoothing the effect of disabling few cache lines, RR caches allow DTM to provide pWCET estimates close to those of fault-free systems despite of faults (see Table 8.2).

Note that while results for LRU caches are deterministic, this is not the case for RR caches, so we report the average execution time across 1,000 runs for each benchmark and configuration for RR caches. In all cases the standard deviation of the execution times for RR is below 0.5%.

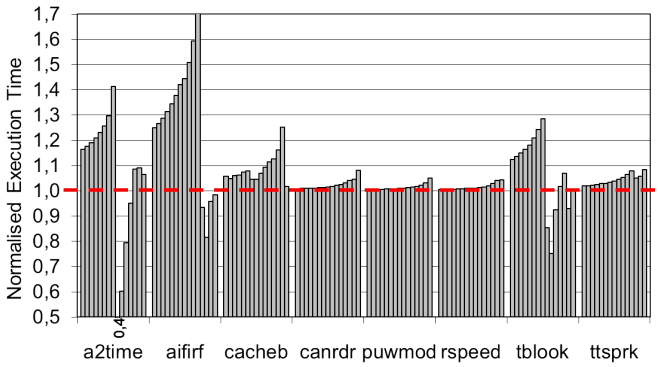


Figure 8.6: Normalised execution time of random-replacement caches with respect to LRU ones.

The increase in terms of average execution time is very similar to the increase in pWCET. For instance, average execution time increases by 28% and 14% for *a2time* and *tblook* respectively when decreasing the number of available cache lines from 64 down to 60. For $p(bit)_f = 10^{-5}$, where 4 cache lines are disabled from both DL1 and IL1, *a2time* and *tblook* respective pWCET estimates grow by 25% and 15%.

Figure 8.6 compares RR caches versus LRU ones showing that PTA-analysability and graceful performance degradation do not come at the expense of a significant performance drop. The figure depicts for each benchmark the slowdown/speedup for each cache size (from 128 to 16 cache lines from left to right for each benchmark). Values higher than 1 indicate RR slowdowns (above the red line) and speedups otherwise. For instance, *a2time* slowdown of RR caches versus LRU ones grows from 16% to 41% when moving from 128 to 72 cache lines; however, when moving to 64 cache lines there is a 60% speedup that decreases as we keep decreasing cache size. Overall, RR execution time across benchmarks and cache sizes is only 7% higher than that of LRU caches.

To summarise, the proposed platform allows delivering probabilistically time-analysable fault-tolerant CRTES whose average execution time is comparable to that of conventional systems.

8.6 Fault-Aware WCET Estimation: Expanding DTM

In the previous Sections we presented and evaluated DTM on degraded hardware due to permanent faults targeting caches. We considered neither the timing impact of error detection, correction, diagnosis and reconfiguration (DCDR) actions, nor

the timing impact of transient faults.

In this and next Sections we propose and evaluate a holistic approach to deal with the timing impact of error DCDR as well as degraded hardware due to both, increasingly frequent permanent and transient faults, to provide strong timing guarantees on a more complex, 4-core multicore system.

8.6.1 Bounding Timing Effects of DCDR

Faults are assumed independent across them as they can be typically modeled as random events. This implies that technology and operating conditions will determine the actual probabilities of bits to experience permanent ($PPbit_{faulty}$) and transient faults ($PTbit_{faulty}$). Next we describe the timing effects of the different actions to deal with faults.

Error Detection and Correction

As mentioned before, the most convenient error detection and correction methods for caches consist of using coding techniques such as, for instance, single error correction double error detection (SECDED). SECDED can be implement correcting data before they are delivered, so that cache read accesses are slower. Alternatively, one could deliver data uncorrected and check for errors offline. If no error occurs cache read latency is effectively decreased. Conversely, on an error, a mechanism is needed to rollback any operation that used uncorrected data, as for instance flushing the pipeline and restarting the execution from the load operation. Since error correction is expected to occur with extremely low probability per instruction, offline correction is recommended. Note that (i) flushing the pipeline is doable since, typically, the error is detected one cycle after data have been delivered, and (ii) during such process corrected data can be stored back in cache to remove wrong data. This last step works correctly for transient errors, but may not solve the problem for permanent faults, as discussed in following sections.

Overall, error detection and correction impact execution time, and that impact is independent of whether the fault producing the error was permanent or transient. Furthermore, the impact of such error in timing is typically constant as error detection, correction and pipeline flushing are expected to have a fixed latency ($Cost_{recover}$).

Transient faults may affect multiple bit cells at once (multiple bit upsets or MBU for short). If an MBU affects multiple bits protected with the same code (e.g., SECDED code), it may be uncorrectable. By interleaving bits protected with different codes (e.g., bits from independently protected words in a SECDED-protected cache line), an MBU affecting b bits becomes b single-bit upsets (SBU), which can be corrected with SECDED.

Fault Diagnosis

Once an error has been detected, it is of prominent importance determining whether it was caused due to a transient or a permanent fault. In the former case, error correction suffices to get rid of the error. In the latter case, however, one should disable the faulty cache line to avoid further errors. Re-reading data after writing back the correct value to diagnose whether the fault was transient or permanent is not a good option since permanent faults typically manifest as intermittent faults whose frequency increases with degradation. Thus, likely we would re-read a correct value despite of a permanent fault.

Instead, in order to deal with error diagnosis we use a similar approach to that in [Abella *et al.* (2008)], where a small table is set up tracking the cache line id (set and way) of the last lines experiencing an error together with a counter tracking how many errors have occurred during a given time period. For instance, one could set up a 2-entry table whose contents are reset every 1,000,000 cycles ($ResetInt = 1000000 \times cycle\ time$, where the cycle time is in seconds), and use a threshold of 3 errors ($ErrorTh = 3$) so that if a cache line reports 3 errors during 1,000,000 cycles it is assumed to have a permanent fault. The selection of the particular values to be used – entries, interval duration and error threshold – are beyond the scope of this Chapter, but this mechanism is almost insensitive to them given that transient faults occur in random locations and infrequently, so they never trigger the deactivation of a cache line. Since fault diagnosis occurs in parallel to regular operation, it has no effect on timing.

Reconfiguration

As is was the case for DTM, if a cache line has been regarded as faulty, it is disabled, using a *Usable Bit* (UB). When setting the UB of a cache line, its contents are discarded or written back to memory if dirty. Therefore, it is required to consider such timing effect at analysis time.

8.6.2 DCDR effect on WCET estimates

After determining how to upper-bound the maximum delay introduced by each DCDR action, next we analyze how many of them need to be considered and how this impacts WCET estimates. Time is typically 'partitioned' in CRTES used in avionics, space and automotive domains. Partitions are implemented in the form of windows, usually in the order of milliseconds, that bound the time a task is allocated to be executed. Based on the time window assigned to a task and its WCET in a fault-free hardware it is easy to compute the probability that the application experiences a number of transient and permanent faults during the

execution of a program without violating its window. Such number, even if fault rates are high, will likely be up to 1 transient ($N_{tran} = 1$) and 1 permanent fault ($N_{perm} = 1$). This is so because the probability of higher fault counts are typically largely below the highest safety level thresholds (e.g. 10^{-9}).

Once the number of transient and permanent faults is known, it is needed to account for the number of error detection and correction actions those faults can trigger:

- **Transient faults**. A transient fault can trigger up to b detection/correction actions if MBUs of up to b bits can occur with non-negligible probability.
- **Permanent faults**. Based on the diagnosis mechanism described before, 3 errors in the interval will imply that the fault is diagnosed as permanent and the cache line disabled, thus avoiding further errors. Hence, the maximum number of errors a permanent fault can potentially trigger during the execution of a program is up to 2 errors per diagnosis interval and 3 errors in the last interval.

Error counts and the upper-bound fixed time required per detection and correction action are then used to upper-bound the impact of detecting and correcting errors in a program ($pWCET_{detect+correct}$) given a maximum time partition duration ($TP_{duration}$).

$$pWCET_{detect+correct} = \left(N_{tran} \cdot b + N_{perm} \cdot \left(\left\lceil \frac{TP_{duration}}{ResetInt} \right\rceil \cdot (ErrorTh - 1) + 1\right)\right) \cdot Cost_{recover} \tag{8.7}$$

If a fault is permanent, the cache line is deactivated and, potentially, dirty line contents are written back. The simplest way to do this without having to consider the probability of a dirty line to be evicted and still obtain sound pWCET estimates consists of performing the writeback operation when a cache line is marked as faulty regardless of whether its contents are dirty. In this way, whether a line is dirty or not has no effect on reconfiguration timing. Still, writeback operations may take different latencies depending on the memory controller arbitration. As the number of permanent faults showing up during a time partition is extremely low, one can simply use an upper-bound to such latency ($Cost_{eviction}$) taking into account the worst arbitration delay for random permutations (up to 2 times the number of contender cores) [Jalle *et al.* (2014)] and the worst memory latency to upper-bound eviction delays. This cost is incurred for each permanent fault ($pWCET_{eviction}$), and must be used to increase the pWCET estimate.

$$pWCET_{eviction} = N_{perm} \cdot Cost_{recover} \tag{8.8}$$

In summary, the pWCET estimate obtained with MBPTA needs to be augmented with $pWCET_{detect+correct}$ and $pWCET_{eviction}$ to account for the instant timing effects of DCDR actions.

8.6.3 Degraded Operation due to Permanent Faults

As described before, DTM can account for the impact of degradation due to permanent faults. So far we have considered only first level caches. However, DTM can also be used for UL2 caches. Multicores using cache partitioning [Paolieri *et al.* (2009a)] in the shared UL2 caches can be easily considered by assuming that each program has a UL2 cache whose size matches its partition as faults will occur with the same faulty bit probability across the whole shared cache[1]. Note that cache partitioning removes any kind of inter-task interference, thus making execution time and WCET independent of the co-runners. Therefore, one only needs to determine the number of faults to be considered in each cache independently, reconfigure the hardware accordingly and collect execution time measurements, as explained before for DTM.

It is worth nothing that those measurements already capture effects such as the impact of inclusion (e.g., if DL1 is inclusive with UL2), as addresses and faults location in cache is random and so the timing observed at analysis time corresponds to that occurring at deployment once the maximum number of permanent faults has been experienced.

Therefore, DTM can be applied in complex cache hierarchies without needing any type of specific adaptation.

8.6.4 Hardware Requirements

Our approach towards considering the potential impact of faults at analysis time in PTA-compliant hardware platforms requires the following hardware support.

1. Error detection and correction methods in caches. Such support generally exists in caches in the form of, for instance, SECDED coding.

2. Per-cache diagnosis support. Each cache memory needs some support to track error location and frequency. As explained before, very small tables are needed (e.g., 2-entry tables with few bits per entry to track set and way number, and error counts) and a counter to decide when to reset those tables.

3. A *UB* bit per cache line to disable faulty lines. Although such bits must be hardened to prevent them from being faulty, thus increasing their size,

[1] Effects related to cache coherence are not considered in our analysis and left as future work.

their cost is negligible given that cache line size is typically in the order of hundreds of bits.

4. Per-cache registers to set up how many permanent faults must be enforced by the hardware. As such fault counts are typically small (typically 1 or 2 faults), 4 bits per counter will suffice.

5. Per-cache support to randomly disable a number of cache lines. If this support is regarded as expensive, one could devise a mechanism allowing the software (i.e. the OS) to randomly choose the lines to be disabled and then disable them with specific instructions that indicate in which cache set of which cache memory a line is to be disabled.

Overall, hardware modifications required are minor and most of them already needed for functional correctness. For instance, Intel Pellston technology (i.e. used in Montecito processor) already implements some sort of mechanism for (1), (2) and (3) above [McNairy & Mayfield (2005)]. Thus, we regard the cost of enabling fault-aware MBPTA as negligibly small. Still, the low-level implementation details are far beyond the scope of this Chapter.

8.6.5 Other Hardware Resources

In general, other hardware resources such as adders, register files, decode logic, etc. may not be disabled without compromising the functionality of the processor. Functional correctness can still be achieved if proper means are in place. For instance, register files can be SECDED-protected, adders may work with residue coding and pipeline flushing on an error, etc. Considering the effect in timing of error DCDR for those features can be done analogously as it is proposed for cache memories. Regarding reconfiguration, typically permanent faults can be tolerated by decreasing operating frequency while keeping operating voltage so that those circuits that used to operate correctly but fail to do it anymore due to degradation, can still operate correctly with extended cycle time. In this case, considering the effect of degraded hardware on pWCET estimates requires determining analytically how much operating frequency may need to be decreased during processor lifetime and obtain execution time measurements for MBPTA at such lower frequency.

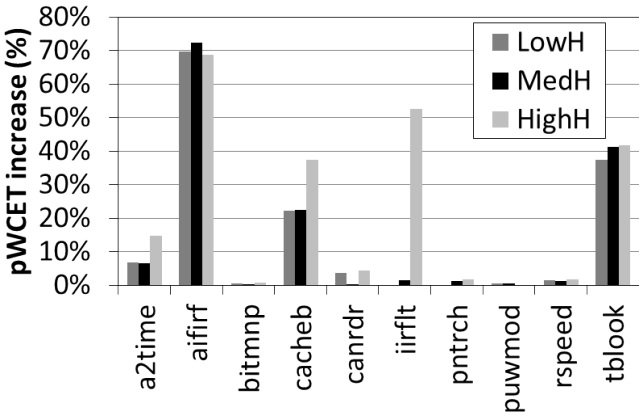
8.7 Evaluation

Execution times have been collected with the already explained SoCLib simulator.

We consider a 4-core multicore, in which cores have private L1 cores and they are connected with a bus with random permutations arbitration [Jalle *et al.* (2014)].

	LowH	MedH	HighH
DL1, IL1	2	4	7
UL2	5	13	49
DTLB, ITLB	1	1	2

(a)



(b)

Figure 8.7: (a) Maximum number of faulty lines expected for a target yield of 1 faulty part per million (ppm); (b) pWCET increase with respect to the fault-free case.

The shared second level cache is partitioned across cores by means of bankization [Paolieri *et al.* (2009a)]. We consider a hit/miss latency of 1 cycle to DL1, IL1, DTLB, ITLB. The bus latency is 2 cycles, once access it is granted to a core. UL2 hit/miss latency is 3 cycles and memory latency (once access it is granted to a request) is 28 cycles. We use the EEMBC Autobench benchmark suite [Poovey *et al.* (2009)].

We consider the following fault rate scenarios:

- Low-harsh (LowH): 10^{-7} permanent faulty bit rate ($PPbit_{faulty}$) and 10^{-4} transient faults per second ($PTbit_{faulty}$) being the operating frequency 500MHz.
- Medium-harsh (MedH): $PPbit_{faulty} = 10^{-6}$ and $PTbit_{faulty} = 10^{-3}$.
- High-harsh (HighH): $PPbit_{faulty} = 10^{-5}$ and $PTbit_{faulty} = 10^{-2}$.

For instance, in the *MedH* scenario we expect 1 bit every 1,000,000 bits become permanently faulty during the processor lifetime, and to experience one transient or permanent fault every 17 minutes. Note that permanent and transient fault rates that we use are similar or higher than actual ones for existing technology [Guertin & White (2010),Wilkerson *et al.* (2008)], thus potentially increasing the negative impact of our approach in pWCET estimates. As shown next, this impact is still low despite of those high fault rates.

The number of cache lines to be disabled for each cache in each scenario is shown in Figure 8.7(a). Figure 8.7(b) shows the pWCET estimate increase obtained with MBPTA of the system considering DCDR impact of errors as well as

degraded operation due to permanent faults with respect to those pWCET estimates obtained in the fault-free system. It can be seen that pWCET estimates increase negligibly in most of the cases as programs are little sensitive to the cache space available, so our approach delivers pWCET estimates close to those on a fault-free system, thus proving their tightness, which will lead to an efficient use of the computing resources and so, low power and energy requirements. Only few programs (e.g., `aifirf`) are more sensitive to the cache space available in some of the caches. Note that, eventually, pWCET estimates may be higher even with more cache space available. For instance, `canrdr` has a higher pWCET estimate in the *LowH* scenario than in the *MedH* one. Those pWCET estimates are trustworthy, but random effects may make that some higher execution times with low probability of occurrence are actually observed and so pWCET estimates become less tight. Further, notice that our approach is not particularly sensitive to the particular cache setup as our setup already includes tiny caches (DTLB, ITLB), medium-size (DL1, IL1) and large caches (UL2) and pWCET estimates increase is low despite of that.

Although not evaluated, the diagnosis mechanism to classify faults into transient or permanent has been shown to have roughly negligible power and area cost, and has no impact on execution time (and so in pWCET estimates) [Abella *et al.* (2008)].

Overall, the proposed platform allows delivering probabilistically time-analyzable fault-tolerant CRTES whose pWCET estimates are close to those of fault-free CRTES.

8.8 Related Work

Timing analysis of systems equipped with cache memories is a serious challenge even for fault-free systems. The impact of caches on WCET has been extensively studied by the research community [Ferdinand *et al.* (2001),Mueller (2000),Reineke *et al.* (2007)], but those techniques show very limited scalability and prevent the adoption of increasingly complex hardware and software.

Some approaches exist to detect errors and recover while keeping time predictability [Axer *et al.* (2011),Henkel *et al.* (2011)]. Those approaches consider the impact in task scheduling of error detection and recovery, and schedule those activities smartly to prevent deadline misses. Furthermore, how to deal with errors at different layers and how to minimise the cost of error detection and recovery is also considered. Unfortunately, the effect in timing of permanent faults is not addressed, so if some hardware resources need to be disabled or reconfigured due to permanent faults, WCET estimates are no longer valid. Hardy et al. [Hardy *et al.* (2016)] have proposed means to account for degraded hardware together

with STA. However, as explained before, deterministic placement and replacement lead to a state explosion to account for all potential fault maps, and keeping the number of states low is only doable with highly pessimistic assumptions.

There are several hardware solutions to keep time predictability despite of permanent faults. Some of them are based on setting up spares to physically replace faulty entries [Koren & Koren (1998)]. Those solutions are typically expensive due to the redundant resources and the costly fuses required to physically reprogram circuits. Another approach consists of using error detection and correction (EDC) codes [Chen & Hsiao (1984)]. This family of solutions is also costly because permanent faults require using EDC logic on each access (energy overhead) and delaying the delivery of data until EDC logic generates a safe output value. A different approach consists of adding some assist structures to perform a *soft* replacement of faulty entries [Abella *et al.* (2011b),Abella *et al.* (2011c)]. This has been proposed for cache memories where victim caches, eviction buffers or similar cache-assist structures are conveniently modified to replace faulty entries while keeping time predictability as needed for WCET analysis, but some extra redundancy and design complexity is introduced.

Alternatively to conventional timing analysis, probabilistic timing analysis (PTA) shows promising results [Cazorla *et al.* (2013),Cucu-Grosjean *et al.* (2012)] for complex platforms running complex applications [Wartel *et al.* (2013)]. Those approaches rely on platforms providing some properties so that execution time variations caused by the hardware itself depend solely on random events. This is achieved, for instance, by using random-replacement caches.

Some existing embedded processors already implement random-replacement policies [Cobham Gaisler (2017),ARM (2006)]. Randomised caches in high-performance processors were first proposed in [Schlansker *et al.* (1993)] to remove pathological cases produced by the systematic cache misses generated in bad strides. To do so, authors used a pseudo-random hash function to randomise addresses into cache sets, developing an analytic approach to determine cache performance. Other cache designs have attempted to remove cache conflicts by changing the placement function [González *et al.* (1997)] and/or combining several placement functions for different cache ways [Seznec & Bodin (1993)]. However, those caches still produce deterministic conflicts across addresses. Recently, direct-mapped and set-associative cache designs implementing random placement have been proposed [Kosmidis *et al.* (2013a)]. Those designs have been successfully proven to fit PTA needs.

Unfortunately, the advent of nanotechnologies poses serious challenges to perform timing analysis while keeping analysis costs low and WCET estimates for applications low, safe and tight. To the best of our knowledge this Chapter proposes the first methodology, DTM, allowing CRTES to provide those low, safe and

tight WCET estimates needed in the CRTES arena despite of faults.

8.9 Conclusions

This Chapter addresses functional and timing correctness in a holistic way by proposing the *Degraded Test Mode (DTM)*, a method to use probabilistically-analysable fault-tolerant hardware designs in combination with MBPTA techniques. The proposed platform delivers (i) probabilistically time-analysable fault-tolerant CRTES (ii) whose WCET is low, trustworthy and tight, and (iii) whose average execution time is comparable to that of conventional systems. Therefore, CRTES can be implemented on top of unreliable hardware despite of faults, thus increasing CRTES performance and reducing their costs.

Chapter 9

Conclusion and Future Work

9.1 Thesis conclusions

Critical Real-Time Embedded Systems (CRTES) need to satisfy *timing correctness* as well as functional correctness. Advanced hardware features, which used to be a promising approach in high-end processors, are effective in providing average case performance, but in CRTES, where it is imperative deriving *trustworthy* Worst-Case Execution Time estimates, they need to be carefully considered. When considering those features, conventional timing analysis techniques either need to make many pessimistic assumptions (static approaches) or their confidence becomes almost impossible to assess (measurement-based approaches), which may easily challenge the usability of those advanced hardware features. Probabilistic Timing Analysis (PTA), and especially its measurement-based variant (MBPTA), has emerged recently as a promising approach to deal with increasingly complex hardware features. However, it has been assessed only on simple single-core processors and small multicores. In this thesis we have proved its applicability on more complex and high-performance processors.

In this thesis we have accomplished to provide tighter WCET estimates in complex multi- and many-core systems, by (1) reducing the contention in shared resources and (2) exploiting the properties of time-randomised cache memories. The main approach in reducing the contention that needs to be accounted for to obtain trustworthy and tight WCET estimates builds upon the observation that contention in shared resources can be handled in a probabilistic way. Therefore, those sequences of events occurring with a negligible low probability can be discarded, thus and not having to assume the worst possible case every time. The main shared resources – the most influencing resources on time-predictability – that we consider in this Thesis are the interconnection network and the shared last level cache memory.

In particular, in this thesis we have reached the following achievements:

- We have proposed tree-based NoC designs suitable for single-criticality and also mixed-criticality CRTES, for a moderate number of cores (up to 16). Our proposal outperforms bus-based multicores, delivers high average performance and can enable mixed criticalities by either managing different priority levels or bandwidth guarantees.
- We have provided a meaningful comparison between two tree-based multicores and showed the benefits of randomisation in deriving tighter WCET estimates. We have done the per-resource analysis and showed which resources are the ones influencing the most tightness of the WCET estimates.
- For processors with a larger number of cores (up to 36 cores, but potentially more) we show that appropriate probabilistic approaches are highly efficient in dealing with contention in mesh wormhole NoCs. We have proposed two different wNoC setups that are able to provide much better performance guarantees than deterministic approaches, which need to account for systematic pathological cases.
- We have discovered a limitation of the existing MBPTA-compliant bus arbitration when requests have heterogeneous duration, and have proposed a control-flow mechanism in order to achieve fairness across cores. In addition to improve guaranteed performance, we show that implementation costs of the mechanism are affordable.
- We have overcome the limitation of partitioned last level caches in CRTES by proposing a mechanism that can be applied on top of the non-partitioned time-randomised last level caches, thus enabling their use in CRTES. Our evaluation showed increased guaranteed performance w.r.t. solutions based on cache partitioning.
- We have proposed a method that allows using fault-tolerant hardware designs to derive trustworthy and tight WCET estimates when used in conjunction with MBPTA-compliant systems. This approach provides also high average execution time. It builds upon the observation that time-randomized caches make fault location irrelevant, thus reducing the problem to accounting for the right number of faults.

9.2 Future work

Every new generation of processors brings more complex processors accounting for larger core counts. Hence, the future research lines emanating from this thesis are

as follows:

- Clustered manycore processor designs with local memories (local for the cluster). Solutions from this thesis (tree-based NoCs) could be applied locally, within clusters, but new challenges would arise for the cluster-to-cluster communication. Such a hierarchical design is interesting in many domains since each cluster provides some form of spatial isolation that may help hypervisors preserving safety, security and availability.

- In this thesis we have focused on symmetrical cores, but more complex CRTES, e.g. multi-sensor, multi-actuator, there can be heterogeneous performance needs, and therefore heterogeneous bandwidth allocation, arbitration, cores and resources. For instance, a mixture of different core types, such as that of ARM big.LITTLE architecture, may be needed in some systems. Also, using GPUs, cryptographic units or other accelerators may be needed in future CRTES. Thus, heterogeneous systems need also to be studied in the future.

- Advanced NoCs features, such as dynamic virtual channel allocation and adaptive routing, despite being extensively used in the high-performance domain, have been traditionally discarded for hard real-time systems due to the difficulties and/or pessimism they introduce when analyzing their timing behavior. Since we showed great performance improvements for wNoCs, we believe that the probabilistic approach may also be very helpful in enabling those features.

References

Abella, J., Chaparro, P., Vera, X., Carretero, J. & Gonzalez, A. (2008). On-line failure detection and confinement in caches. In *On-Line Testing Symposium, 2008. IOLTS '08. 14th IEEE International*. 129, 134

Abella, J., Carretero, J., Chaparro, P., Vera, X. & Gonzalez, A. (2009). Low vccmin fault-tolerant cache with highly predictable performance. In *MICRO*. 110, 114

Abella, J., Cazorla, F., Quiñones, E., Gizopoulos, D., Grasset, A., Yehia, S., Bonnot, P., Mariani, R. & Bernat, G. (2011a). Towards improved survivability in safety-critical systems. In *International On-Line Testing Symposium (IOLTS)*. 109

Abella, J., Quiñones, E., Cazorla, F., Sazeides, Y. & M.Valero (2011b). RVC-Based time-predictable faulty caches for safety-critical systems. In *IOLTS*. 110, 118, 135

Abella, J., Quinones, E., Cazorla, F., Sazeides, Y. & Valero, M. (2011c). RVC: A mechanism for time-analyzable real-time processors with faulty caches. In *HiPEAC Conference*. 24, 110, 118, 135

Abella, J., Quiñones, E., Vardanega, T. & Cazorla, F. (2013). Measurement-based probabilistic timing analysis and i.i.d property. White Paper. http://www.proartis-project.eu/publications/MBPTA-white-paper. 16, 102

Abella, J., Hardy, D., Puaut, I., Quiñones, E. & Cazorla, F.J. (2014). On the comparison of deterministic and probabilistic WCET estimation techniques. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, 266–275. 48

Abella, J., Hernández, C., Quiñones, E., Cazorla, F.J., Conmy, P.R., Azkarate-askasua, M., Perez, J., Mezzetti, E. & Vardanega, T.

(2015). Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 39–48, IEEE. 5, 13

Agirre, I., Azkarate-askasua, M., Hernández, C., Abella, J., Perez, J., Vardanega, T. & Cazorla, F.J. (2015). IEC-61508 SIL 3 compliant pseudo-random number generators for probabilistic timing analysis. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, 677–684. 63, 82

Altmeyer, S. & Davis, R.I. (2014). On the correctness, optimality and precision of static probabilistic timing analysis. In *DATE*. 14, 90

ARINC (1997). *Specification 651: Design Guide for Integrated Modular Avionics*. Aeronautical Radio, Inc. 32, 100

ARM (1999). *AMBA Bus Specification. http://www.arm.com/products/system-ip/amba/amba-open-specifications.php*. 82, 83

ARM (2006). *Cortex-R4 and Cortex-R4F Technical Reference Manual*. 135

AUTOSAR (2006). *Technical Overview V2.0.1*. 100

Axer, P., Sebastian, M. & Ernst, R. (2011). Reliability analysis for MPSoCs with mixed-critical, hard real-time constraints. In *CODES+ISSS*. 134

Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M. & Marwedel, P. (2002). Scratchpad memory: design alternative for cache on-chip memory in embedded systems. In *CODES*. 19

Benini, L., Flamand, E., Fuin, D. & Melpignano, D. (2012). P2012: Building an ecosystem for a scalable, modular and high-efficiency embedded computing accelerator. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '12. 22, 32

Bernat, G., Colin, A. & Petters, S.M. (2002). Wcet analysis of probabilistic hard real-time system. In *RTSS*, 279–288, IEEE Computer Society. 14

Bogdan, P., Kas, M., Marculescu, R. & Mutlu, O. (2010). Quale: A quantum-leap inspired model for non-stationary analysis of noc traffic in chip multi-processors. In *2010 Fourth ACM/IEEE International Symposium on Networks-on-Chip*, 241–248. 73

Bowman, K., Duvall, S. & Meindl, J. (2002). Impact of die-to-die and within-die parameter fluctuations on the maximum clock frequency distribution for gigascale integration. *IEEE Journal of Solid-State Circuits*, **2**. 23, 111

Bradley, J. (1968). *Distribution-Free Statistical Tests*. Prentice-Hall. 29

Bui, D. & Lee, E.A. (2012). Composable flexible real-time packet scheduling for networks on-chip. Tech. rep., University of Berkeley. 88

Campoy, M., Ivars, A.P. & Mataix, J.V.B. (2001). Static use of locking caches in multitask preemptive real-time systems. In *In Proceedings of IEEE/IEE Real-Time Embedded Systems Workshop (Satellite of the IEEE Real-Time Systems Symposium*. 19

Cazorla, F.J., Quiones, E., Vardanega, T., Cucu, L., Abella, J., Bernat, G. & Triquet, B. (2010). PROARTIS EU-FP7 project deliverables, http://www.proartis-project.eu/deliverables. 26

Cazorla, F.J., Gioiosa, R., Fernandez, M. & Quiñones, E. (2012). Multicore os benchmarks. Tech. Rep. 4000102623, European Space Agency. 7

Cazorla, F.J., Quiñones, E., Vardanega, T., Cucu, L., Triquet, B., Bernat, G., Berger, E., Abella, J., Wartel, F., Houston, M., Santinelli, L., Kosmidis, L., Lo, C. & Maxim, D. (2013). Proartis: Probabilistically analysable real-time systems. *ACM Transactions on Embedded Computing Systems*. x, 5, 6, 14, 90, 135

Chattopadhyay, S., Roychoudhury, A. & Mitra, T. (2010). Modeling shared cache and bus in multi-cores for timing analysis. In *13th International Workshop on Software and Compilers for Embedded Systems (SCOPES 2010)*, St. Goar, Germany. 92

Chen, C. & Hsiao, M. (1984). Error-correcting codes for semiconductor memory applications: A state of the art review. *IBM Journal of Research and Development*, **28**, 124–134. 110, 135

Chiou, D., Jain, P., Devadas, S. & Rudolph, L. (2000). Dynamic cache partitioning via columnization. In *DAC*, Los Angeles, CA, USA. 19, 49

Clarke, P. (2011). Automotive chip content growing fast, says gartner. In *http://www.eetimes.com/electronics-news/4207377/Automotive-chip-content-growing-fast*. 2

Cobham Gaisler (2017). *LEON 4 Processor*. http://www.gaisler.com/index.php/products/processors/leon4. 8, 18, 81, 135

Coles, S. (2001). *An Introduction to Statistical Modeling of Extreme Values*. Springer. 16

Cucu-Grosjean, L., Santinelli, L., Houston, M., Lo, C., Vardanega, T., Kosmidis, L., Abella, J., Mezzetti, E., Quiñones, E. & Cazorla, F.J. (2012). Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems, ECRTS*. 5, 6, 14, 15, 29, 30, 76, 90, 96, 119, 121, 122, 124, 135

Cullmann, C., Ferdinand1, C., Gebhard, G., Grund, D., Maiza, C., Reineke, J., Triquet, B. & Wilhelm, R. (2010). Predictability considerations in the design of multi-core embedded systems. In *ERTS*. 89

D. Siewiorek, P.N. (2006). Fault tolerant architectures for space and avionics. In *joint IARP/IEEE-RAS/EURON and IFIP 10.4 Workshop on Dependability in Robotics and Autonomous Systems*. 3

DeMicheli, G., Leblebici, Y., Gijs, M. & Voros, J. (2009). *Nanosystems Design and Technology*. Springer. 23

Dubois, F., Cano, J., Coppola, M., Flich, J. & Petrot, F. (2011). Spidergon stnoc design flow. In *Proceedings of the Fifth ACM/IEEE International Symposium on Networks-on-Chip*, NOCS '11, 267–268, ACM. 26

Edelin, G. (2009). Embedded systems at THALES: the artemis challenges for an industrial group. In *presentation at the ARTIST Summer School in Europe*. 3

Federal Aviation Administration (FAA) (2014). Airborne electronic hardware. CAST-32 position paper. multi-core processors. 89

Feller, W. (1966). *An introduction to Probability Theory and Its Applications*. John Willer and Sons. 15, 29, 118

Ferdinand, C. & Wilhelm, R. (1999). Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time System*, **XVII**, 131–181. 92

Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S. & Wilhelm, R. (2001). Reliable and precise wcet determination for a real-life processor. 134

Fernández, M., Gioiosa, R., Quiñones, E., Fossati, L., Zulianello, M. & Cazorla, F.J. (2012). 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. 77

Ferrandiz, T., Frances, F. & Fraboul, C. (2012). A sensitivity analysis of two worst-case delay computation methods for spacewire networks. In *Euromicro Conference on Real-Time Systems (ECRTS), Pise, 11/07/2012-13/07/2012*. 73

Flich, J. & Bertozzi, D. (2010). *Designing Network On-Chip Architectures in the Nanoscale Era*. Chapman & Hall/CRC. 26

Francisco J. Cazorla, E.Q., Tulio Vardanega & Abella, J. (2013). Upper-bounding program execution time with extreme value theory. In *WCET Workshop*. 15

Freescale (1997). *PowerPC 750 Microprocessor*. Freescale. 25

FreeScale (2012). *P4080 QorIQ Integrated Multicore Communication Processor Family Reference Manual. Rev 1*. 8, 18

Garrido, M. & Diebolt, J. (2000). The ET test, a goodness-of-fit test for the distribution tail. In *MMR*. 29

Gizopoulos, D., Psarakis, M., Adve, S., Ramachandran, P., Hari, S., Sorin, D., A.Meixner, A.Biswas & X.Vera (2011). Architectures for on-line error detection and recovery in multicore processors. In *DATE*. 24

González, A., Valero, M., Topham, N. & Parcerisa, J.M. (1997). Eliminating cache conflict misses through XOR-based placement functions. In *Proceedings of the 11th International Conference on Supercomputing*, ICS '97, 76–83. 135

Goossens, K., Dielissen, J. & Radulescu, A. (2005). Æthereal network on chip: Concepts, architectures, and implementations. *IEEE Design & Test of Computers*, **22**, 414–421. 71, 88

Gorgues, M., Xiang, D., Flich, J., Yu, Z. & Duato, J. (2014). Achieving balanced buffer utilization with a proper co-design of flow control and routing algorithm. In *Eighth IEEE/ACM International Symposium on Networks-on-Chip, NoCS 2014, Ferrara, Italy, September 17-19, 2014*, 25–32. 26

Guertin, S. & White, M. (2010). CMOS reliability challenges the future of commercial digital electronics and NASA. In *NEPP Electronic Technology Workshop*. xi, 23, 24, 110, 112, 113, 133

Hansen, J., Hissam, S. & Moreno, G.A. (2009). Statistical-based wcet estimation and validation. In *WCET Workshop*. 14

Hardy, D. & Puaut, I. (2008). WCET analysis of multi-level non-inclusive set-associative instruction caches. In *Proceedings of the 29th IEEE Real-Time Systems Symposium, RTSS 2008, Barcelona, Spain, 30 November - 3 December 2008*, 456–466. 19, 92

Hardy, D. & Puaut, I. (2013). Static probabilistic worst case execution time estimation for architectures with faulty instruction caches. In *RTNS*. 24

Hardy, D., Puaut, I. & Sazeides, Y. (2016). Probabilistic wcet estimation in presence of hardware for mitigating the impact of permanent faults. In *2016 Design, Automation Test in Europe Conference Exhibition (DATE)*, 91–96. 110, 134

Henkel, J., Bauer, L., Becker, J., Bringmann, O., Brinkschulte, U., Chakraborty, S., Engel, M., Ernst, R., Härtig, H., Hedrich, L., Herkersdorf, A., Kapitza, R., Lohmann, D., Marwedel, P., Platzner, M., Rosenstiel, W., Schlichtmann, U., Spinczyk, O., Tahoori, M.B., Teich, J., Wehn, N. & Wunderlich, H. (2011). Design and architectures for dependable embedded systems. In *Proceedings of the 9th International Conference on Hardware/Software Codesign and System Synthesis, CODES+ISSS 2011, part of ESWeek '11 Seventh Embedded Systems Week, Taipei, Taiwan, 9-14 October, 2011*. 134

Hernández, C., Abella, J., Cazorla, F., Andersson, J. & Gianarro, A. (2015). Towards making a LEON3 multicore compatible with probabilistic timing analysis. In *DASIA*. 10, 15, 81, 82, 83

Hernández, C., Abella, J., Gianarro, A., Andersson, J. & Cazorla, F.J. (2016). Random modulo: a new processor cache design for real-time critical systems. In *Proceedings of the 53rd Annual Design Automation Conference, DAC 2016, Austin, TX, USA, June 5-9, 2016*, 29:1–29:6. 20, 50

Inc., N. (2014). *The NanGate 45nm Open Cell Library*. 45

Infineon (2012). AURIX - TriCore datasheet. highly integrated and performance optimized 32-bit microcontrollers for automotive and industrial applications. 8, 89

J. Owens (2015). Delphi automotive, the design of innovation that drives tomorrow. Keynote talk. In Design Automation Conference (DAC). 3

Jain, S.K. & Agarwal, P. (2006). A low leakage and snm free sram cell design in deep sub micron cmos technology. In *VLSID*. 111

Jalle, J., Abella, J., Quiñones, E., Fossati, L., Zulianello, M. & Cazorla, F.J. (2013a). Deconstructing bus access control policies for real-time multicores. In *8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013, Porto, Portugal, June 19-21, 2013*, 31–38. 21

Jalle, J., Abella, J., Quiñones, E., Fossati, L., Zulianello, M. & Cazorla, F.J. (2013b). Deconstructing bus access control policies for real-time multicores. In *8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013, Porto, Portugal, June 19-21, 2013*, 31–38. 75, 76

Jalle, J., Kosmidis, L., Abella, J., Quiñones, E. & Cazorla, F.J. (2014). Bus designs for time-probabilistic multicore processors. In *Design, Automation & Test in Europe Conference & Exhibition, DATE 2014, Dresden, Germany, March 24-28, 2014*, 1–6. 8, 21, 22, 31, 32, 33, 34, 50, 56, 62, 73, 75, 77, 80, 82, 83, 88, 101, 130, 132

Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S. & Roychoudhury, A. (2011). Bus-aware multicore wcet analysis through tdma offset bounds. In *Proceedings of the 2011 23rd Euromicro Conference on Real-Time Systems*, ECRTS '11, 3–12. 21

Kelter, T., Falk, H., Marwedel, P., Chattopadhyay, S. & Roychoudhury, A. (2014). Static analysis of multi-core TDMA resource arbitration delays. *Real-Time Systems*, **50**, 185–229. 75

Kim, B., Kim, J., Hong, S.J. & Lee, S. (1998). A real-time communication method for wormhole switching networks. In *1998 International Conference on Parallel Processing (ICPP '98), 10-14 August 1998, Minneapolis, Minnesota, USA, Proceedings*. 59

Kim, H., Kandhalu, A. & Rajkumar, R. (2013). A coordinated approach for practical os-level cache management in multi-core real-time systems. In *25th Euromicro Conference on Real-Time Systems, (ECRTS 2013)*, Paris, France. 9, 90, 91, 92

Koren, I. & Koren, Z. (1998). Defect tolerance in vlsi circuits: techniques and yield analysis. *Proceedings of the IEEE*, **86**, 1819–1838. 110, 135

Kosmidis, L., Abella, J., Quiñones, E. & Cazorla, F.J. (2013a). A cache design for probabilistically analysable real-time systems. In *Design, Automation and Test in Europe, DATE 13, Grenoble, France, March 18-22, 2013*, 513–518. 18, 20, 38, 50, 65, 91, 95, 96, 98, 100, 101, 113, 118, 122, 125, 135

Kosmidis, L., Abella, J., Quiñones, E. & Cazorla, F.J. (2013b). Multi-level unified caches for probabilistically time analysable real-time systems. In *Proceedings of the IEEE 34th Real-Time Systems Symposium, RTSS 2013, Vancouver, BC, Canada, December 3-6, 2013*, 360–371. 18, 20, 38, 65, 102

Kosmidis, L., Abella, J., Wartel, F., Quiñones, E., Colin, A. & Cazorla, F.J. (2014). PUB: path upper-bounding for measurement-based probabilistic timing analysis. In *26th Euromicro Conference on Real-Time Systems, ECRTS 2014, Madrid, Spain, July 8-11, 2014*, 276–287. 15

Kotz, S. & Nadarajah, S. (2000). *Extreme value distributions: theory and applications*. World Scientific. 6, 14

Lahiri, K., Raghunathan, A. & Lakshminarayana, G. (2001). Lotterybus: a new high-performance communication architecture for system-on-chip designs. In *Proceedings of the 38th Design Automation Conference, DAC 2001,*. 21, 73, 75, 77

Law, S. & Bate, I. (2016). Achieving appropriate test coverage for reliable measurement-based timing analysis. In *ECRTS*. 12

Le Boudec, J.Y. & Thiran, P. (2001). *Network calculus: a theory of deterministic queuing systems for the internet*. Springer-Verlag. 71

Lee, S. (2003). Real-time wormhole channels. *Journal Of Parallel And Distributed Computing*, **63**, 299–311. 72

Lesage, B., Hardy, D. & Puaut, I. (2009). WCET analysis of multi-level set-associative data caches. *WCET Workshop*. 19, 92

Liedtke, J., Hartig, H. & Hohmuth, M. (1997). OS-controlled cache predictability for real-time systems. In *Real-Time and Embedded Technology and Applications Symp. (RTAS)*. 9, 19, 90, 91, 92

Lu, Z., Yao, Y. & Jiang, Y. (2014). Towards stochastic delay bound analysis for network-on-chip. In *Eighth IEEE/ACM International Symposium on Networks-on-Chip (NoCS)*, 64–71. 72, 73

Lyons, R. & Vanderkulk, W. (1962). The use of triple modular redundancy to improve computer reliability. *IBM Journal of Research and Development*, **6**, 200–209. 110

Marsaglia, G. & Zaman, A. (1991). A new class of random number generators. *Annals of Applied Probability*, **1**, 462–480. 99

Massengill, L., Bhuva, B., Holman, W., Alles, M. & Loveless, T. (2012). Technology scaling and soft error reliability. In *IEEE International Reliability Physics Symposium (IRPS)*. 24

McNairy, C. & Mayfield, J. (2005). Montecito error protection and mitigation. In *HPCRI*. 114, 132

Mezzetti, E. & Vardanega, T. (2011). On the industrial fitness of wcet analysis. In *WCET Workshop*. 12

Mezzetti, E., Ziccardi, M., Vardanega, T., Abella, J., Quiñones, E. & Cazorla, F.J. (2015). Randomized caches can be pretty useful to hard real-time systems. *LITES*, **2**, 01:1–01:10. 19

Millberg, M., Nilsson, E., Thid, R., Kumar, S. & Jantsch, A. (2004). The nostrum backbone-a communication protocol stack for networks on chip. In *IEEE VLSI Design*, 693–696. 88

Milutinovic, S., Mezzetti, E., Abella, J., Vardanega, T. & Cazorla, F. (2017). On uses of extreme value theory fit for industrial-quality WCET analysis. In *2017 12th IEEE International Symposium on Industrial Embedded Systems (SIES)*. 15

Mueller, F. (1994). Predicting instruction cache behavior. *Language, Compilers and Tools for Real-Time Systems*. 19, 92

Mueller, F. (1995). Compiler support for software-based cache partitioning. In *ACM SIGPLAN Conference on Languages, Compilers and Tools for Embedded Systems (LCTES)*. 9, 19, 90, 91, 92

Mueller, F. (2000). Timing analysis for instruction caches. *Real-Time Systems - Special issue on worst-case execution-time analysis*. 134

NanoC (2010). Nanoc design platform. http://www.nanoc-project.eu. 26, 38, 65

Natale, M.D., Abella, J., Reineke, J., Hamann, A. & Farrall, G. (2016). Predictable system timing – probab(ilistical)ly? In *DAC (panel in automotive track)*. 12

Obermaisser, R., El-Salloum, C., Huber, B. & Kopetz, H. (2009). From a federated to an integrated automotive architecture. In *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*. 1

Panic, M., Rodríguez, G., Quiñones, E., Abella, J. & Cazorla, F.J. (2013). On-chip ring network designs for hard-real time systems. In *21st International Conference on Real-Time Networks and Systems, RTNS 2013, Sophia Antipolis, France, October 17-18, 2013*, 23–32. 31, 88

Panic, M., Quiñones, E., Zaykov, P.G., Hernández, C., Abella, J. & Cazorla, F.J. (2014). Parallel many-core avionics systems. In *2014 International Conference on Embedded Software, EMSOFT 2014, New Delhi, India, October 12-17, 2014*, 26:1–26:10. 22, 32, 33, 34

Panic, M., Abella, J., Hernández, C., Quiñones, E., Ungerer, T. & Cazorla, F.J. (2015). Enabling TDMA arbitration in the context of MBPTA. In *2015 Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015*, 462–469. 21, 73, 80

Panic, M., Hernández, C., Abella, J., Roca, A., Quiñones, E. & Cazorla, F.J. (2016a). Improving performance guarantees in wormhole mesh noc designs. In *2016 Design, Automation & Test in Europe Conference & Exhibition, DATE 2016, Dresden, Germany, March 14-18, 2016*, 1485–1488. 55, 66, 68

Panic, M., Hernández, C., Quiñones, E., Abella, J. & Cazorla, F.J. (2016b). Modeling high-performance wormhole nocs for critical real-time embedded systems. In *2016 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS), Vienna, Austria, April 11-14, 2016*, 267–278. 55, 60, 66, 67, 73, 89

Paolieri, M., Quiñones, E., Cazorla, F.J., Bernat, G. & Valero, M. (2009a). Hardware support for WCET analysis of hard real-time multicore systems. In *Proceedings of the 36th Annual International Symposium on Computer Architecture (ISCA)*. 9, 20, 21, 34, 39, 48, 90, 91, 92, 105, 131, 133

Paolieri, M., Quiñones, E., Cazorla, F.J. & Valero, M. (2009b). *An Analyzable Memory Controller for Hard Real-Time CMPs*. IEEE Embedded Systems Letters. 101, 102

Poovey, J.A., Conte, T.M., Levy, M. & Gal-On, S. (2009). A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, **29**, 18–29. 28, 66, 83, 101, 120, 133

Pouillon, N., Becoulet, A., de Mello, A., Pecheux, F. & Greiner, A. (2009). A generic instruction set simulator api for timed and untimed simulation and debug of mp2-socs. In *Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on*, 116–122, http://www.soclib.fr/trac/dev. 25, 38, 65, 100, 120

Psarras, A., Seitanidis, I., Nicopoulos, C. & Dimitrakopoulos, G. (2015). Phase-NoC: TDM scheduling at the virtual-channel level for efficient network traffic isolation. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition, DATE*. 89

Rahmati, D., Murali, S., Benini, L., Angiolini, F., Micheli, G.D. & Sarbazi-Azad, H. (2013). Computing accurate performance bounds for best effort networks-on-chip. *IEEE Transactions on Computers*. 55, 58, 73, 89

Rapita Systems, L. (2007). *RapiTime: Worst-case execution time analysis. User Guide*. 13

Reineke, J., Grund, D., Berg, C. & Wilhelm, R. (2007). Timing predictability of cache replacement policies. *Real-Time Systems*, **37**, 99–122. 19, 92, 134

Roberts, D., Kim, N. & Mudge, T. (2007). On-chip cache device scaling limits and effective fault repair techniques in future nanoscale technology. In *DSD Euromicro Conference*. 110, 114

Roca, A., Hernández, C., Flich, J., Silla, F. & Duato, J. (2012). Enabling high-performance crossbars through a floorplan-aware design. In *ICPP*. 31, 38, 45

RTCA and EUROCAE (2011). *DO-178C / ED-12C, Software Considerations in Airborne Systems and Equipment Certification*. 1, 5, 14, 35

Salminen, E., Kangas, T., Lahtinen, V., Riihimäki, J., Kuusilinna, K. & Hämäläinen, T.D. (2007). Benchmarking mesh and hierarchical bus networks in system-on-chip context. *J. Syst. Archit.*, **53**. 75

Santinelli, L., Morio, J., Dufour, G. & Jacquemart, D. (2014). On the Sustainability of the Extreme Value Theory for WCET Estimation. In *14th International Workshop on Worst-Case Execution Time Analysis*, vol. 39 of *OpenAccess Series in Informatics (OASIcs)*, 21–30. 16

Schlansker, M., Shaw, R. & Sivaramakrishnan, S. (1993). Randomization and associativity in the design of placement-insensitive caches. *HP Tech Report HPL-93-41*. 135

Schoeberl, M., Brandner, F., Sparsø, J. & Kasapaki, E. (2012). A statically scheduled time-division-multiplexed network-on-chip for real-time systems. In *NOCS*. 31, 88

Schranzhofer, A., Chen, J. & Thiele, L. (2010). Timing analysis for TDMA arbitration in resource sharing systems. In *16th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2010, Stockholm, Sweden, April 12-15, 2010*, 215–224. 21

Seznec, A. & Bodin, F. (1993). Skewed-associative caches. In *PARLE*. 135

Shi, Z. & Burns, A. (2008). Real-time communication analysis for on-chip networks with wormhole switching. In *NoCS*. 71

Suhendra, V., Mitra, T., Roychoudhury, A. & Chen, T. (2005). WCET centric data allocation to scratchpad memory. In *Proceedings of the 26th IEEE Real-Time Systems Symposium (RTSS*. 19

Theiling, H., Ferdinand, C. & Wilhelm, R. (2000). Fast and precise WCET prediction by separated cache and path analyses. *Real-Time Systems*, **18**, 157–179. 92

Thottethodi, M., Lebeck, A.R. & Mukherjee, S.S. (2001). Self-Tuned Congestion Control for Multiprocessor Networks. In *HPCA*. 64

Tschanz, J., Kao, J., Narendra, S., Nair, R., Antoniadis, D., Chandrakasan, A. & De, V. (2002). Adaptive body bias for reducing impacts of die-to-die and within-die parameter variations on microprocessor frequency and leakage. *IEEE Journal of Solid-State Circuits*, **37**. 114

Ungerer, T., Bradatsch, C., Gerdes, M., Kluge, F., Jahr, R., Mische, J., Fernandes, J., Zaykov, P.G., Petrov, Z., Böddeker, B., Kehr, S., Regler, H., Hugl, A., Rochange, C., Ozaktas, H., Cassé, H., Bonenfant, A., Sainrat, P., Broster, I., Lay, N., George, D., Quiñones, E., Panic, M., Abella, J., Cazorla, F.J., Uhrig, S., Rohde, M. & Pyka, A. (2013). parmerasa – multi-core execution of parallelised hard real-time applications supporting analysability. In *DSD*. 35

Vestal, S. (2007). Preemptive scheduling of multi-criticality systems with varying degrees of execution time assurance. In *RTSS*, 239–243. 2, 31

Vittorelli, B. (2004). Automotive systems. where silicon meets the road. **3**, information Quarterly Building Blocks for Embedded Applications. 3

Ward, B., Herman, J., Kenna, C. & Anderson, J. (2013). Making shared caches more predictable on multicore platforms. In *25th Euromicro Conf. on Real-Time Systems, (ECRTS)*. 9, 90, 91, 92

Wartel, F., Kosmidis, L., Lo, C., Triquet, B., Quiñones, E., Abella, J., Gogonel, A., Baldovin, A., Mezzetti, E., Cucu, L., Vardanega, T. & Cazorla, F.J. (2013). Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *8th IEEE International Symposium on Industrial Embedded Systems,SIES*. 15, 96, 135

Wartel et al., F. (2015). Timing analysis of an avionics case study on complex hardware/software platforms. In *18th Design, Automation and Test in Europe Conference (DATE)*. 15

Wassel, H.M.G., Gao, Y., Oberg, J., Huffmire, T., Kastner, R., Chong, F.T. & Sherwood, T. (2013). SurfNoC: A low latency and provably non-interfering approach to secure networks-on-chip. In *The 40th Annual International Symposium on Computer Architecture ISCA*, 583–594. 89

Wenzel, I. (2006). *Measurement-Based Timing Analysis of Superscalar Processors*. Ph.D. thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria. 13

Wenzel, I., Kirner, R., Rieder, B. & Puschner, P.P. (2008). 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*, 430–444. 13

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J. & Stenström, P. (2008). The worst-case execution-time problem—overview of methods and survey of tools. *ACM Trans. Embed. Comput. Syst.*, **7**, 36:1–36:53. 5, 13, 90, 110

Wilkerson, C., Gao, H., Alameldeen, A., Chishti, Z., Khellah, M. & Lu, S.L. (2008). Trading off cache capacity for reliability to enable low voltage operation. In *ISCA*. 110, 114, 133

Yue Qian, Z.L. & Dou, W. (2009). Analysis of worst-case delay bounds for best-effort communication in wormhole networks on chip. In *NoCS*, 44–53. 72

Zhang, W. & Yan, J. (2012). Static timing analysis of shared caches for multicore processors. *Journal of Computing Science and Engineering*, **6**. 92

Ziccardi, M., Mezzetti, E., Vardanega, T., Abella, J. & Cazorla, F.J. (2015). EPC: Extended Path Coverage for Measurement-Based Probabilistic Timing Analysis. In *Real-Time Systems Symposium, 2015 IEEE*, 338–349. 15

Glossary

CRTES Critical Real-Time Embedded Systems

DTM Degraded Test Mode

DCDR Detection, Correction, Diagnosis and Reconfiguration

DTA Deterministic Timing Analysis

EFL Eviction Frequency Limiting

ETP Execution Time Profile

EVT Extreme Value Theory

FIT Failure In Time

FIFO First-In First-Out

IPC Instructions Per Cycle

LLC Last Level Cache

LRU Least Recently Used

LNR Limiting Number of Requests

LFR Limiting Frequency of Requests

MiD Minimum Inter-request Delay

MBPTA Measurement-Based Probabilistic Timing Analysis

NoC Network-On-Chip

PTA Probabilistic Timing Analysis

pWCET Probabilistic Worst-Case Execution Time

RTOS Real-Time Operating System

RR Round-Robin

SECDED Single-error-correction and double-error-detection

SPTA Static Probabilistic Timing Analysis

TDMA Time-Division Multiple Access

TuA Task under Analysis

UBD Upper-Bound Delay

wNoC Wormhole Network-On-Chip

WCD Worst Contention Delay

WCET Worst-case execution time