Smart Hardware Designs for Probabilistically-Analyzable Processor Architectures

Pedro Benedicte

Universitat Politècnica de Catalunya

Computer Architecture Department

PhD thesis

Doctoral programme on Computer Architecture

January, 2022

Smart Hardware Designs for Probabilistically-Analyzable Processor Architectures

Pedro Benedicte
January 2022

Universitat Politècnica de Catalunya
Computer Architecture Department

A Thesis submitted in fulfillment of
the requirements for the degree of
Doctor of Philosophy in Computer Architecture

Advisor: Francisco J. Cazorla, PhD, Barcelona Supercomputing Center
Tutor: Jesús Labarta, PhD, Universitat Politècnica de Catalunya

Abstract

Future Critical Real-Time Embedded Systems (CRTES), like those is planes, cars or trains, require more and more guaranteed performance in order to satisfy the increasing performance demands of advanced complex software features. While increased performance can be achieved by deploying processor techniques currently used in High-Performance Computing (HPC) and mainstream domains, their use challenges the software timing analysis, a necessary step in CRTES' verification and validation. Cache memories are known to have high impact in performance, and in fact, current CRTES include multicores usually with several levels of cache. In this line, this Thesis aims at increasing the guaranteed performance of CRTES by using techniques for caches building upon time randomization and providing probabilistic guarantees of tasks' execution time.

In this Thesis, we first focus on on improving cache placement and replacement to improve guaranteed performance. For placement, different existing policies are explored in a multi-level cache setup, and a solution is reached in which different of those policies are combined. For cache replacement, we analyze a pathological scenario that no cache policy so far accounts and propose several policies that fix this pathological scenario.

For shared caches in multicore we observe that contention is mainly caused by private writes that go through to the shared cache, yet using a pure write-back policy also has its drawbacks. We propose a hybrid approach to mitigate this contention. Building on this solution, the next contribution tackles a problem caused by the need of some reliability mechanisms in CRTES. Implementing reliability close to the processor's core has a significant impact in performance. A look-ahead error detection solution is proposed to greatly mitigate the performance impact.

The next contribution proposes the first hardware prefetcher for CRTES with arbitrary cache hierarchies. Given its speculative nature, prefetchers that have a guaranteed positive impact on performance are difficult to design. We present a framework that provides execution time guarantees and obtains a performance benefit.

Finally, we focus on the impact of timing anomalies in CRTES with caches. For the first time, a definition and taxonomy of timing anomalies is given for Measurement-Based Timing Analysis. Then, we focus on a specific timing anomaly that can happen with caches and provide a solution to account for it in the execution time estimates.

Acknowledgements

Doing a Ph.D. is an odd endeavour, certainly not for the fainthearted. However, as any challenge in life, it would not be possible without the help, one way or another, of other human beings. Thus, this is a small but significant acknowledgement of the group of people that accompanied me during my Ph.D. time.

First and foremost, I want to deeply thank my Thesis advisors: Fran, Jaume and Carles. Any student would be grateful to have just one of them as a mentour, in my case I was lucky to have all three of them. I keep with me many lessons learned, both technical and personal, that they have been kind enough to teach me.

As well as my advisors, the rest of the people of the CAOS research group have given me lots of help over many coffees and *palmeras*: Leonidas, Mikel, Maria, Milos, Mladen, Edu, Enrico, Suzana, Miquel, Javi, Javier, Jeremy, Sergi, Jordi and many others.

I also want to thank my university colleagues that have accompanied me for the almost 10 years that I have spent learning at UPC: David, Albert, Constan and Cristo. I am sure we will cross paths again, likely on industry grounds.

Furthermore, I have to write some lines about my normal (or at least not computer science savvy) friends: Pol, Sandro, Eric, Paula, Anna, Guallarte, Cabezas and Itziar. Thank you for always pushing me to aim for more while reminding me to also enjoy life along the way.

I want to also thank my sister, mostly for putting up with me all this time, but also for all the travels shared together.

I also want to thank the wonderful people of Arm Cambridge that hosted me during my internship there, as well as my advisors in Austin.

Finally, I must thank my parents, not only for their support throught the duration of the Ph.D., but for their patience, teachings and confidence in me during my entire life. Anything good that I can become or achieve in this life will of course be thanks to them.

I am not really good at this sort of nontechical writing, so to summarize: Thanks. Gracias. Gràcies.

Contents

List of Figures

List of Tables

List of Abbreviations

BT Binary Tree. 7, 46–48, 57, 59, 60, 63–66, 68, 69, 139

CRTES Critical Real-Time Embedded Systems. 1, 2, 4–7, 11, 13, 17, 18, 22, 45, 69, 117, 126, 141

ECC Error Correction Codes. 5, 7, 22, 140

FPGA Field-Programmable Gate Array. 24

HDL Hardware Description Language. 8, 24, 25, 134, 141

HPC High Performance Computing. 5, 7, 13, 18

hRP hash Random Placement. 19, 20, 34–42

HWM High-Water Mark. 12

IMA Integrated Modular Avionics. 4

LRU Least Recently Used. 7, 20, 39, 46–48, 52, 57, 58, 62–66, 68–70, 119, 121, 139

MBPTA Measurement-Based Probabilistic Timing Analysis. 3, 5–8, 13, 15–18, 20, 21, 24, 27, 33–35, 37, 39, 41, 43, 45, 48, 52, 53, 57–63, 66–70, 128–132, 135, 137, 139, 140, 142

MBTA Measurement-Based Timing Analysis. 3–6, 8, 13–15, 17, 105, 106, 109, 111, 127–129, 137, 140, 141

MLC Multi-Level Caches. 2, 3, 5, 71

MOD Modulo. 35–42

NMRU Non-Most Recently Used. 7, 46–49, 56–60, 62–66, 68–70, 139

NMRURP Non-Most Recently Used Random Permutations. 45, 46, 49, 54–57, 62–70, 139

pWCET Probabilistic Worst-Case Execution Time. 8, 15, 16, 21, 24, 27, 28, 33, 35, 38–40, 42, 48–50, 63, 65, 67–70, 123, 124, 130, 133, 135, 136

RM Random Modulo. 20, 33–41, 63

RP Random Permutations. 26, 45, 46, 49–57, 61–63, 65, 66, 68–70, 132, 139

RR Random Replacement. 5, 7, 18, 20, 21, 38, 45, 48–51, 53–55, 57, 61–66, 68–70, 119–122, 139, 141

RTOS Real-Time Operating System. 37, 42, 57, 81

SECDED Single Error Correction, Double Error Detection. 22

SLLC Shared Last-Level Cache. 7

SoC System on Chip. 1, 17

STA Static Timing Analysis. 3–6, 8, 13–18, 74, 127–129, 131, 137, 140, 141

WCET Worst-Case Execution Time. 2–4, 6, 11–18, 24, 33, 34, 42, 43, 45, 46, 51, 58, 61, 62, 70, 72, 74, 75, 77, 79, 85–89, 91, 93, 95, 105, 109, 113, 114, 117, 126, 127, 129, 131, 133, 137, 141

Chapter 1

Introduction

1.1 Trends in Critical Real-Time Embedded Systems

Traditionally, the use of computers has been mainly focused on automation of functions in personal computing, office/scientific work, and media consumption. The systems used for these purposes have usually been relatively big and composed of several components (separated CPU, GPU, memory, etc). Some of the most used computing systems are laptops, desktops or servers/mainframes. However, in the last decades a different kind of computing systems have gained relevance in our lives: embedded systems. Such is the omnipresence of embedded systems that they alredy accounted for 98% of the total number of computer systems [113] a few years back. In the past, embedded systems have been related to low-performance, cheap, and function-specific devices. In recent years, however, embedded systems have started implementing more complex functions requiring significant performance on general purpose processors [63]. This change has been driven by the increased functionality and demand of these systems, such as for instance the increase of automation in almost all industries. Furthermore, with the introduction of System on Chip (SoC) in mobile devices, the line between embedded systems and dedicated computers (such as personal computers) has blurred.

A subset of embedded systems are used in industries such as automotive, aerospace, space or railway. As an example, in a car lots of functions previously performed by the driver are now automatically performed by the car. These functions can go from relatively simple ones such as automatically managing the air conditioning so the car always has the same temperature, to complex ones such as automatically steering the car. Depending on the criticality of the function performed, special considerations related to correct and timely operation have to be taken into account to ensure safety. These systems are referred to as Critical Real-Time Embedded Systems (CRTES) or real-time systems. The additional requirements of CRTES can be mainly summarized into two: functional correctness and timing correctness. Functional correctness implies that the system behaves as it was specified. Timing correctness, which is targeted in this Thesis, requires the overal system functionalty to behave in a timely manner. This carries that each task executes before a specific time, called deadline.

Guaranteeing that deadlines are met, in turn, builds on deriving estimates to the Worst-Case Execution Time (WCET) of critical real-time functionalities.

In terms of growth, the computer industry has experienced a significant growth since its conception, possibly more than any other industry. As an example, if the automotive industry had grown at the same pace of processor's performance since 1971, the fastest car would now travel at a tenth of the speed of light [149]. The CRTES industry has also grown, driven by the increasing critical real-time features demands of industries such as aerospace, automotive, rail or space, which require higher levels of performance than those provided by simple microcontrollers used traditionally [35, 121] Taking automotive as representative domain, on-board software in cars already comprises hundreds of millions of lines of code [40], with its performance requirements expected to rise by two orders of magnitude [2] by 2024. It is widely accepted that timely executing those software functionalities will rely on processors comprising high-performance features. Those performance levels can only be achieved using powerful processors implementing high-performance features including cache memories, multicore processors and/or prefetchers. Unfortunately, these components usually rely on speculation and hard to predict dynamic behavior, which challenge the estimation of WCET.

Overall, the need for increased performance in CRTES along with the challenging timing analyzable requirements, calls for new designs and methodologies that can synergistically provide both.

1.2 Challenges in the Memory Subsystem of CRTES

The trends described in the previous section challenge CRTES. In this Thesis we focus on providing high performance while increasing time analyzability (with special focus on timing anomalies and time composability) and reliability. We address the combined challenge specifically at the memory subsystem level which is known to have a huge impact on average and predictable performance.

1.2.1 High Performance

Using powerful processors that implement high-performance features provides high average performance but challenge the estimation of WCET, a fundamental step for timing validation and verification and hence for assessing the correct timing behavior of CRTES [157]. In this line, using complex hardware in CRTES requires increased performance guarantees (i.e. reduced WCET estimates) – and not just increased average performance as needed in the mainstream market.

A prominent feature that improves average performance are cache memories with high-performance processors, which are already ubiquitously deploying several levels of cache (e.g. IBM POWER 9, Intel Core i7-based systems, and the ARM A Series). This is also true of the latest embedded processors (NXP P4080 [119], Xilinx Zynq UltraScale+ [163], NXP PowerQUICC III [136]...). This emanates from the significant impact that Multi-Level Caches (MLC) have on overall system performance.

However, MLC design is delicate, because it involves high complexity when dealing with coherence, inclusion, placement and replacement policies; and can also affect key metrics like cycle time, energy/power (and hence temperature), reliability and design complexity.

1.2.2 Time Analyzability

The main challenge when bringing HPC hardware features to the real-time domain is their impact on guaranteed performance, (i.e. the ability to produce tight WCET estimates). The process of obtaining timing guarantees for a piece of software is called software timing analysis or simply timing analysis. Several families of techniques exist, with the most prominent ones being Static Timing Analysis (STA) and Measurement-Based Timing Analysis (MBTA), each one approaching timing analysis from a different perspective.

Traditionally, STA has been the reference technique, and given the longevity of many real-time systems there are still lots of systems nowadays that have been analyzed using this technique. STA derives the WCET without actually executing the application. It does so with models and abstract representations of the software and the timing behavior of the underlying hardware. However, as hardware complexity increases, STA suffers from some limitations to the point that it might result in overly pessimistic (and hence not useful) timings [156]. For instance, it has been shown that adding a new hardware feature to improve average performance can result in a worse WCET estimate than the same system without the feature enabled [152]. Furthermore, STA requires in-depth information about the hardware used. It is normal for manufacturers to not provide this information for confidentiality reasons, and even sometimes they do not know all the possible timing states that the system can have.

In this context, MBTA has proliferated as the most used timing analysis technique nowadays. MBTA executes the task on the target hardware under stressful scenarios and collects execution task measurements. However, MBTA also has its constraints and challenges [4]. An important challenge to MBTA is the representativity of the analysis runs. In a nutshell, MBTA obtains timings from what are called analysis runs. These runs are performed in the same system that will later be used in deployment. These analysis runs must be representative of the worst possible scenario that the system can undergo at deployment. However, obtaining the worst-case in the analysis runs is not trivial. Furthermore, not all the hardware conditions and states are accessible to the programmer, so in some cases it may even be impossible to set up the specific worst conditions.

In the recent years, Measurement-Based Probabilistic Timing Analysis (MBPTA) has emerged in order to address this representativity problem. While it is also based on the measurement-based paradigm, it moves the problem of representativity from the engineer to the system. The explanation and considerations of STA, MBTA and MBPTA are explained in depth in Section 2.1.

All in all, time analyzing a task for a real-time system is becoming more and more challenging as hardware complexity continues to increase. While classic methodologies such as STA were sufficient for old systems, new complex systems that have increased performance require new techniques to obtain timing guarantees.

1.2.2.1 Timing anomalies

As a major source of complexity in timing verification, modern processors are generally prone to timing anomalies [105, 78, 158], a well-known phenomenon causing that local worst-cases are not guaranteed to lead to the global worst-case.

Hence, dealing with timing anomalies is required to enable and support the analysis of complex computing platforms. This applies to both STA, for which timing anomalies have been deeply analyzed [78, 158], and MBTA [155], for which, instead, timing anomalies have been totally neglected so far.

The challenge with timing anomalies is related to the Measurement-Based techniques. As MBTA is increasingly used in CRTES, potential timing anomalies must be identified and handled correctly.

1.2.2.2 Time composability

The process of designing and integrating a CRTES has several stages. Usually, different vendors provide the software that will run in the system. First, they develop the software. Then, they time analyze their software on the system where it will run. However, due to factors such as development time constraints or confidentiality, only their software is being tested on the system, when in deployment it will run with other tasks. Then, at the end of the process, software from different vendors will be integrated in the same system. However, since the tasks have been only time analyzed in isolation, the timings obtained will not be valid when running them together. This is due to potential effects that can emerge from contention.

Time composability eases the development of CRTES since it allows deriving timing estimates in early design stages with assurance that they remain valid as different software components, which are developed independently, are incrementally integrated. Usually time composable WCET estimates introduce some overheads, since some pessimistic assumptions are made to guarantee that the timing estimates hold when running with other tasks. Time composibility is becoming a fundamental property in increasingly-complex multi-provider software projects in integrated systems like Integrated Modular Avionics (IMA) [1, 161] in the avionics domain or AUTOSAR in the automotive domain [62].

1.2.3 Hardware Reliability

One of the trends that has allowed processors to keep increasing performance at Moore's law pace has been shrinking transistor size. It has allowed the integration of more transistors in the same die area, which has several advantages such as allowing more complex designs, operate at higher frequencies or reducing power dissipation, amongst others. However, the reduction in transistor size impacts the reliability of processors. This is so because a smaller transistor size usually holds less charge, and a smaller charge is typically easier to flip.

While this is an important problem for High Performance Computing (HPC), it is specially critical for CRTES. Critical systems must undergo a strict certification process to provide evidence that hardware failure rates are below specific thresholds set in applicable safety standards, e.g. ISO26262 [77] in cars. Critical systems include safety mechanisms for fault tolerance to ensure low-enough acceptable failure rates.

Specifically, the sensitivity of caches to errors (faults) in CRTES is a challenge for system designers. A bit flip in a cache could cause an unexpected result on the executing task, leading to a non-safe scenario. Thus, some sort of reliability mechanism is required in caches to make them certifiable in CRTES.

1.3 Motivation

The challenges described in Section 1.2 open the door to several research lines. With focus on cache memories, the main focus of this Thesis is to enable the use HPC features in CRTES under the umbrella of MBPTA. This can have a positive impact in four specific areas: improving guaranteed performance and time analyzability in caches, reducing multicore contention while taking into consideration reliability and guaranteed performance constraints, improving the guaranteed performance with the addition of a prefetcher to the memory subsystem, and improving the confidence in timing analysis by tackling timing anomalies in MBTA.

- **Caches guaranteed performance and time analyzability.** Caches are used in CRTES in order to improve guaranteed performance. MBPTA can be used as a timing analysis methodology to overcome several limitations of STA and MBTA with caches. In order to enable MBPTA, time randomized caches have been introduced, both using random placement and Random Replacement (RR). However, more complex systems implement several cache levels to improve performance. The use of random placement policies in MLC challenges the time composability of such systems, as well as their performance. In addition, existing RR policies have several pathological scenarios that can significantly reduce the guaranteed performance of the system.
- **Multicore contention.** Multicore architectures in CRTES present some performance challenges [126]. Mainly, the potential contention on the shared resources comes from the bus or the shared last-level cache. The cache write policy of these systems has a big impact on the contention they suffer. Furthermore, the write policy used in these systems affects the reliability mechanisms that must be put in place to pass safety regulations. Specifically, when allowing the possibility of having dirty data in L1 caches, the addition of Error Correction Codes (ECC) mechanisms close to the core can negatively affect performance. Alternatively, avoiding dirty data in L1 caches requires sending abundant data to L2 caches, which increases performance degradation due to contention.
- **Hardware prefetcher.** Given the time guarantees necessary for CRTES, it is challenging to use certain HPC features, mostly because in general they make use of speculation. A component that has not been brought to CRTES is the

Table 1.1: Contributions, focus of work and publications.

Topic	**Subtopic**	**Focus**	**Publications**
1. Caches	Placement	High-Performance Time Analyzability Time Composability	DATE 2018
2. Caches	Replacement	High-Performance Time Analyzability	SAC 2018 JSA 2019
3. Multicore	Contention	High-Performance Time Analyzability	ECRTS 2018
4. Multicore	Reliability	High-Performance Time Analyzability Reliability	DATE 2019
5. Prefetching		High-Performance, Time Analyzability	Submitted to JSA 2022
6. Timing anomalies		Time Analyzability Timing Anomalies	ASPDAC 2019

hardware prefetcher, both for data and instructions, while providing guaranteed times. Prefetchers are critical to hide memory latencies for cold and capacity misses.

- **Timing anomalies in MBTA.** The consideration of timing anomalies is critical for the confidence on the timing estimates. Because of this, timing anomalies have been deeply studied in the context of STA. However, there is neither a definition of timing anomalies for MBTA systems, nor solutions to account for them in WCET estimates.

1.4 Contributions

This Thesis advances the hardware designs of CRTES to increase their guaranteed performance while guaranteeing the specific needs of CRTES. In particular, the main focus of the work is the memory subsystem, specifically the caches and prefetcher, which have shown to have a high performance impact in HPC systems. Since CRTES inherit these high-performance designs (multi-level caches, multicore systems, prefetchers...), several challenges arise to make sure that these designs are suitable for and can cover the needs of CRTES' needs.

The contributions in this Thesis are divided into six major themes. As a summary, Table 1.1 shows the topic and subtopic of each contribution, as well as the areas where each of them focus and the publications related to each one of them (in more detail in Section 1.6).

1. The first contribution of this Thesis is about bringing MBPTA compliant cache placement policies to multi-level cache systems. The main challenges when tack-

ling this problem are obtaining good guaranteed performance, maintaining time composability and managing the implementation complexity. In this contribution we analyze several combinations of placement policies for multilevel caches building on top of existing random placement policies for single-level caches. We quantify in a simulator which is the best one in terms of guaranteed performance that satisfies the time composability constraints and implement that one in an FPGA multicore prototype.

2. The second contribution also relates to random caches that enable MBPTA, but in this case we focus on the replacement policies. RR, the standard random policy used in MBPTA, suffers from pathological scenarios that decrease its guaranteed performance. First, we identify those scenarios, which are also suffered by most non-random replacement policies, such as Least Recently Used (LRU), Non-Most Recently Used (NMRU) and Binary Tree (BT). Then, we propose two different policies that are also time randomized but do not suffer from those pathological cases. Finally, we quantify the impact on guaranteed performance of the improved policies in a simulator.
3. The next contribution is about write policies in multicore systems. Assuming a multicore system with several levels of cache, whose last level is shared amongst all cores, the contention caused by the shared resources (interconnect, Shared Last-Level Cache (SLLC)) can increase significantly the guaranteed execution time. The write policy of private caches (write-through, write-back) has a huge impact on the number of messages sent to the SLLC, as well as on the complexity of the coherency protocol and energy consumption. We analyze both write policies and conclude that the optimal is a hybrid policy that behaves as write-though with shared data and as write-back with private data. We design this policy along with a simple solution to discriminate private and shared data, and implement it in a simulator, obtaining a significant improvement in terms of guaranteed performance, energy and complexity.
4. The fourth contribution tackles on an open challenge of the previous contribution. One drawback of the proposed hybrid write solution is that it requires error correction in the L1 cache, since it may have dirty data. Implementing ECC so close to the core can have a significant impact in performance due to the latency to check and generate codes. First, we analyze the potential performance impact of putting ECC in the L1 data cache. Then, we propose several incremental improvements culminating in our proposal: a look-ahead mechanism that allows us to mitigate the timing impact of ECC. Finally, we implement this solution in a simulator and quantify the improvement in the guaranteed execution time increase as a result of using our proposed technique.
5. The next contribution of the Thesis focuses on reducing cold and capacity misses. To this end, a component commonly used in the HPC domain is brought to CRTES: the hardware prefetcher. Since providing timing guarantees with HPC components is challenging due to their speculative nature, we propose a formal framework that allows hardware prefetchers to be implemented. We simulate the proposed framework and obtain improvements in performance

guarantees.

6. Our final contribution tackles the topic of timing anomalies in MBTA. While timing anomalies have been deeply analyzed for STA, they have not been studied for MBTA. We define timing anomalies in the context of MBTA and classify the different types of timing anomalies. Then, we focus on a specific timing anomaly than can happen in a real system, and propose a solution to properly handle it.

1.5 Thesis Organization

The thesis is organized as follows:

- **Chapter 1 - Introduction**. It introduces the main topics addressed, explains the different contributions and lists the publications that have been produced as a result of the work done in this Thesis.
- **Chapter 2 - Background**. It presents the background required to understand the rest of the Thesis. It explains the basics of Timing Analysis as well as the details of some specific TA families. The next topic tackled is caches in the real-time systems domain, as well as other challenges such as contention, reliability and prefetching.
- **Chapter 3 - Experimental Setup**. It describes the experimental setup starting with the overall methodology used in the Thesis. Then, the reference processor used, the simulator infrastructure and the Hardware Description Language (HDL) setup are described. Then, the MBPTA tools and methodology that allow for Probabilistic Worst-Case Execution Time (pWCET) computation are discussed. The Chapter concludes explaining the different benchmarks used.
- **Chapters 4 to 9 - Contributions**. Each contribution of this Thesis is presented as an individual Chapter. The organization each of these Chapters is similar: the first section introduces the topic and the challenges. The next section (or sections if needed) attack the main problem, explaining the issues and our proposed solution. Afterwards, the setup and evaluation are explained. The following section collects the related work in that topic. Finally, at the end of each section, the conclusions of the contribution are summarized.

  - **Chapter 4 - Cache placement policies**
  - **Chapter 5 - Cache replacement policies**
  - **Chapter 6 - Cache write policies**
  - **Chapter 7 - Cache redundancy**
  - **Chapter 8 - Prefetching**
  - **Chapter 9 - Timing anomalies**

- **Chapter 10 - Conclusions** Finally, Chapter 10 concludes the work done in this Thesis and discusses the potential work that could be inspired in the future.

1.6 List of Publications

As a product of the work done in this Thesis, 6 publications have been made (5 in international conferences and 1 in a journal) and 1 more publication has been submitted to a journal.

1. **Design and Integration of Hierarchical-Placement Multi-level Caches for Real-Time Systems** [22]
   Pedro Benedicte, Carles Hernandez, Jaume Abella, Francisco J. Cazorla
   Design, Automation and Test in Europe (DATE), 2018
   DOI: 10.23919/DATE.2018.8342052
2. **RPR: A Random Replacement Policy with Limited Pathological Replacements** [24]
   Pedro Benedicte, Carles Hernandez, Jaume Abella, Francisco J. Cazorla
   ACM Symposium on Applied Computing (SAC), 2018
   DOI: 10.1145/3167132.3167197
3. **HWP: Hardware Support to Reconcile Cache Energy, Complexity, Performance and WCET estimates in Multicore Real-Time Systems** [23]
   Pedro Benedicte, Carles Hernandez, Jaume Abella, Francisco J. Cazorla
   EUROMICRO Conference on Real-Time Systems (ECRTS), 2018
   DOI: 10.4230/LIPIcs.ECRTS.2018.3
4. **Improving Time-Randomized Cache Design** [27]
   Pedro Benedicte, Carles Hernandez, Jaume Abella, Francisco J. Cazorla
   5th BSC Severo Ochoa Doctoral Symposium, 2018
5. **Towards Limiting the Impact of Timing Anomalies in Complex Real-Time Processors** [21]
   Pedro Benedicte, Jaume Abella, Carles Hernandez, Enrico Mezzetti, Francisco J. Cazorla
   Asia and South Pacific Design Automation Conference (ASPDAC), 2019
   DOI: 10.1145/3287624.3287655
6. **LAEC: Look-Ahead Error Correction Codes in Embedded Processors L1 Data Cache** [25]
   Pedro Benedicte, Carles Hernandez, Jaume Abella, Francisco J. Cazorla
   Design, Automation and Test in Europe (DATE), 2019
   DOI: 10.23919/DATE.2019.8714877
7. **Locality-aware Cache Random Replacement Policies** [26]
   Pedro Benedicte, Carles Hernandez, Jaume Abella, Francisco J. Cazorla
   Journal of Systems Architecture (JSA), 2019
   DOI: 10.1016/j.sysarc.2018.12.007
8. **A Formal Framework and Its Instantiation for the Design of Time-Predictable Hardware Prefetchers**
   Pedro Benedicte, Carles Hernandez, Jaume Abella, Francisco J. Cazorla
   Submitted to Journal of Systems Architecture (JSA)

Additional publications, although not directly related to the contributions in this Thesis, have been published in the same domain and with related objectives.

8. **Modelling the confidence of timing analysis for time randomised caches** [28]
   Pedro Benedicte, Leonidas Kosmidis, Jaume Abella, Francisco J. Cazorla
   IEEE Symposium on Industrial Embedded Systems (SIES), 2016
   DOI: 10.1109/SIES.2016.7509421
9. **A confidence assessment of WCET estimates for software time randomized caches** [29]
   Pedro Benedicte, Leonidas Kosmidis, Jaume Abella, Francisco J. Cazorla
   IEEE International Conference on Industrial Informatics (INDIN), 2016
   DOI: 10.1109/INDIN.2016.7819140

Chapter 2

Background

2.1 Timing Analysis

Responsiveness of Critical Real-Time Embedded Systems (CRTES) is essential to their correct operation. As such, these systems must react within a limited time called *deadline* (for instance a steering system may need to react within 50ms [90]) to prevent any timing related miss-behavior.

In order to satisfy this time constraint, for each task an upper-bound of the execution time is estimated during the development of the software and its integration in the platform to assess it against the task deadline. This upper-bound is usually called Worst-Case Execution Time (WCET) estimate and it is derived via a process called timing analysis.

The first step to deriving WCET estimates is identifying the possible factors that can affect the execution time of a program. For instance, sources of jitter at a hardware level can be complex and hard to predict in specific features of the processor (e.g. cache memories). Undocumented system functionalities, which are not controlled by the end user, can also lead to unexpected timing behavior across runs of the same task.

At the software level, another possible source of time variation can be the (data) inputs of the program. The inputs can have an effect on the control flow of the program, executing different parts of the software. Moreover, different inputs can result in different execution times even when the same instructions are executed. For example, some floating point or mathematical operations such as multiplication or division can take a different number of execution cycles depending on the input operands.

An additional potential source of time variation is resource sharing. For performance and cost reasons, some components are shared amongst different programs in multicores, such as the computing core (e.g. multithreaded cores), memory hierarchy or interconnects. When different programs access the same resource at the same time, it can result in contention in the shared resource. If this happens, it will likely affect the execution time of the programs sharing the resource, and variability will emanate from the contention caused by other programs and low level arbitration effects.

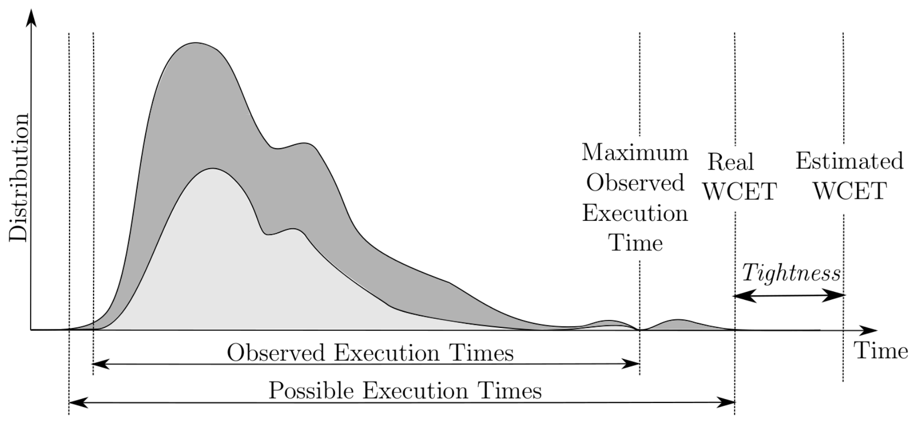


Figure 2.1: Execution time distributions (from [157])

Figure 2.1 illustrates in dark grey the distribution of all the possible execution times. The Figure shows in the horizontal axis the different potential execution times, and in the vertical axis the frequency of occurrence of that specific execution time. We observe that, typically, most of the potential execution times fall under the same range of values; while the highest possible execution time values happen only a small number of times.

Obtaining the complete distribution of all the possible execution times of a program (the dark grey distribution in the Figure) is a challenging task, since an exhaustive search of all the possible factors that affect the execution time would need to be done. A more realistic approach is to analyze only a subset of all the potential execution times. In Figure 2.1 this subset is the distribution of possible execution times in light grey.

2.1.1 WCET

The computation of the WCET is a complex endeavor. Taking into account that it is commonly unfeasible to obtain the distribution of all possible execution times, three different relevant execution times are defined (as seen in Figure 2.1):

- Maximum Observed Execution Time, or High-Water Mark (HWM) is the largest execution time observed. The method for obtaining this value can change depending on the timing analysis technique used.
- Real WCET is the maximum execution time that the specific program under analysis can produce in the system operation stage.
- Estimated WCET is the execution time that results from the timing analysis and it is intended to upper-bound the real WCET.

The difference between the estimated WCET and the real WCET is defined as the tightness. The bigger this difference is, the more overestimation has been incurred when computing the estimated WCET. Another important parameter when obtaining the estimated WCET is that the value is trustworthy. Trustworthiness of the estimated WCET is related to the evidence that the real WCET does not exceed the estimated WCET. Thus, an important objective of timing analysis is to obtain WCET estimates as tight as possible while being trustworthy.

This presents an interesting differentiation with respect to High Performance Computing (HPC) design goals. In HPC, the main objective is to improve average performance [69], which in Figure 2.1 would result in shifting the bulk of the execution time distribution to the left, even if that meant adding a small number of cases that increase the WCET. While usually not being the primary objective, improving average performance can also be beneficial for CRTES. Specifically, in mixed criticality systems, where tasks that have critical time requirements and others that do not are executed on the same platform, potentially simultaneously. In improving the average execution time, the programs that do not have critical constraints improve their performance.

Deriving WCET estimation is challenging and has evolved with the complexity of the hardware and software in CRTES. We first introduct Static Timing Analysis (STA) and Measurement-Based Timing Analysis (MBTA); and then cover Measurement-Based Probabilistic Timing Analysis (MBPTA) a more recent timing analysis methodology based in MBTA that has rapidly evolved in the last decade to tackle the limitations of STA and MBTA.

2.1.2 STA

STA is a family of timing analysis techniques based on deriving the WCET through analytic methods, without actually executing the application on the hardware. Usually mathematical models and abstract representations are used to achieve this purpose. STA has been used in relatively simple CRTES. The trustworthiness of this technique is based on the input information provided, like timing information about the execution of instructions and flow facts. As the complexity of both hardware and software increases in future more ambitious CRTES, the use of STA has problems to guarantee that this analytic methods accurately describe hardware and software [111, 4].

Usually, STA consists of mainly two phases: low-level and high-level analyses [157]. In the low-level analysis an accurate model of the architecture of the process is designed. In order to derive WCET estimates, the potential states that the hardware can have at any given point are modelled [123]. Thus, the trustworthiness of this technique is directly related with the fidelity with which the model resembles the actual hardware. While this is an achievable task with simple hardware, it becomes unattainable with complex modern hardware features. Furthermore, it relies on hardware manufacturers providing detailed information about the implementation and timings of their designs, which is not often available due to the confidential and competitive nature of the market. Even in the cases that this information is available, it is detailed in lengthy complex documents, that are usually accompanied by subsequent errata documents, which reflect the unreliability of this information [4].

The high-level analysis focuses on the program structure by means of its control-flow graph. All the different paths that can be taken are analysed. However, given the complexity of programs and the fact that the control path can change significantly with different inputs, it is sometimes highly challenging or unfeasible to provide a sound analysis.

While STA is still used in some specific domains, with the complexity increase of both software and hardware it is increasingly challenging to apply STA to derive safe WCET estimates.

2.1.3 MBTA

The most extended timing analysis practice nowadays is MBTA [157]. MBTA relies on collecting task's execution time measurements on the target hardware during the system analysis (or design) phase under different stressing conditions with guarantees that those conditions capture the worst scenarios that can arise during operation. MBTA can be used for the timing analysis of the highest-criticality tasks, as it has been shown for avionics software [99].

In order to better understand the timing analysis methodology of MBTA, two stages need to be defined: analysis and operation. First, in the analysis or development stage the system designers, software developers, systems integrators and others obtain WCET estimates. During this stage, engineers can have extensive access to software and hardware to make several tests. Then, the operation stage is when the system is already in operation in the real world performing the task it was designed to perform. The WCET estimates derived in the analysis stage need to hold during the operation stage. In order for the resulting WCET estimates to be trustworthy, the conditions that the system will undertake in operation must be carefully understood and taken into consideration in the analysis stage. This is one of the most important challenges of MBTA, and is commonly referred to as *representativeness*.

Another key component of MBTA is the collection of measurements. Software instrumentation can be used to include instructions that perform this collection. However, they modify the code and can result in a different timing behavior if deployed without such instrumentation. A solution to this problem is to deploy the instrumented code or a nop-instruction-based code with the same memory layout: it will have a small overhead due to the instrumentation/nop code but the WCET estimates computed at analysis will not suffer the probe effect [50].

Another potential challenge with MBTA is the gap between the Maximum Observed Execution Time and the Real WCET (as seen in Figure 2.1). Obtaining the Real WCET in the analysis runs is a challenging task. Not only it requires information about the hardware in order to set up this specific case, but even with this information sometimes setting up this case is unfeasible or highly challenging. This happens since there may be no specific knobs in the hardware to set certain parameters, such as forcing several cores to make a request to a shared bus at the same instant. In order to solve this problem, the Maximum Observed Execution Time is increased by a safety margin. This margin usually depends on the domain and criticality, and tries to account for uncertainties on the timing of the system. As an example, a 20% margin may be used in the avionics industry [154]. This margin has been deemed to be acceptable due to past experience: systems with this margin have

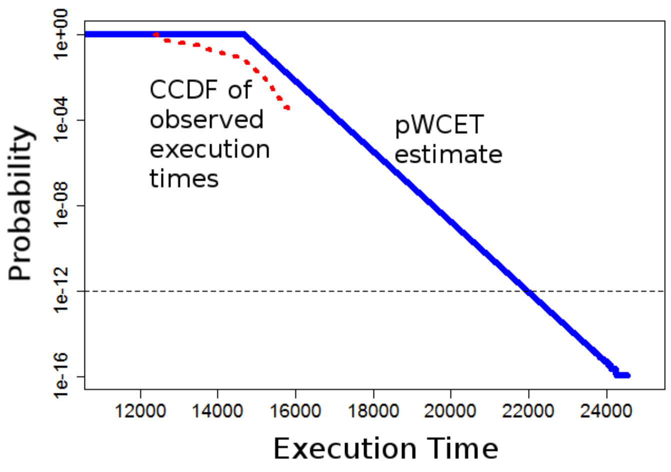


Figure 2.2: pWCET estimate curve derived from the CCDF measurements

never experienced any overruns. This has important implications, since some works point out that this 20% is not sufficient for multicore processors [56].

Finally, an additional challenge with this methodology is the number of runs that need to be performed at analysis to provide a trustworthy WCET estimate. Ideally, an exhaustive number of runs exploring all the possible combinations of both inputs and hardware states should be ran. In reality, it is unfeasible, so it is the task of the engineer to design and execute a set of experiments that are deemed representative of the worst possible timing outcomes. Hence, the trustworthiness of the resulting WCET estimates depends on the design of these tests.

2.1.4 MBPTA

In order to address the main challenges of timing analysis that suffer STA and MBTA techniques, a new paradigm has arisen in the last decade: MBPTA [39, 48]. Instead of providing a single execution time as a result of different executions of a program (given the same program inputs and hardware state), MBPTA benefits from injecting some changed in the platform to handle representativity, which we describe later in this section. Some of these changes relate to adding randomization to the timing behavior of hard-to-predict hardware components which cause variability in tasks' execution time that is bounded not with single WCET estimate, but a Probabilistic Worst-Case Execution Time (pWCET) curve.

MBPTA builds on a set of execution time measurements taken during the system analysis phase, whose Complementary Cumulative Distribution Function (CCDF) is shown with a dashed red line in Figure 2.2 for a given synthetic example. Those measurements are passed as input to Extreme Value Theory (EVT) [132], a statistical technique to estimate an upper-bound distribution tails.

Another feature of MBPTA is that it controls how execution time measurements are collected so they capture those conditions that lead to higher or equal execution times than those during system operation. EVT requires that the execution times meet several statistical properties related to the degree of independence and identical distribution of the random variable (execution times) modeled. Also, MBPTA controls whether execution times can be modeled with an exponential tail, which is the most convenient distribution for pWCET estimates of real-time programs [45, 5].

MBPTA delivers a pWCET distribution, often depicted as a CCDF, so that for each particular execution time value we obtain an upper-bound probability with which it can be exceeded (see blue solid line in Figure 2.2). Therefore, the pWCET estimate is the value such that its upper-bound exceedance probability can be regarded as irrelevant in relation to acceptable failure rates in the corresponding safety standards (e.g. ISO26262 in automotive [77]). For instance, as shown in Figure 2.2, the probability of the program to take 22,000 cycles or longer is below an exceedance probability of 10^{-12} per run.

Interestingly, EVT is able to predict the probabilities for combinations of events that have not occurred simultaneously in any of the observations in the sample. For instance, if those observations correspond to the execution time when experiencing between 10 and 20 cache misses, EVT can predict the probabilities for execution times caused by a larger number of misses (e.g. caused by 50, 100 or 1,000 misses) [7].

In order for MBPTA to provide sound estimates, it requires the sources of execution time variability (jitter) to be either upper-bounded (during analysis) or time-randomized (during both analysis and operation). For instance, latencies due to values operated in variable-latency units are typically upper-bounded, and cache placement is typically randomized. By applying these techniques to the different sources of jitter of the processor, MBPTA relieves the user from controlling during testing low level aspects of those resources causing jitter, for which the end user may lack means to determine and enforce their worst timing behavior at analysis.

MBPTA is a mature technology that has been successfully assessed with case studies in the automotive, railway, space, and avionics domains [153, 57]. MBPTA builds on the underlying (complex) platform having certain properties in its timing behavior as a means to facilitate the analysis of software timing.

2.1.5 Timing Anomalies

When providing guarantees that the WCET estimates are trustworthy, a phenomenon that deserves special attention is that of timing anomalies. A high-level definition of timing anomalies, from the perspective of timing analysis, is given [78] as those cases where a local worst-case does not lead to the global worst-case. An illustrative timing anomaly is shown in Figure 2.3 where instructions in each row execute serially due to data dependencies (i.e. E consumes some data produced by A), and C and E use the same resource. We see that, by experiencing a longer latency for A (local effect), the overall execution time decreases (global effect).

For STA, timing anomalies jeopardize the formal reliability of the approach. STA is in fact forced to resort to some form of abstraction to be able to model all possible inputs and hardware states, as shown later in Chapter 9. Abstractions in turn introduce non-determinism in the model, where an abstract state can have multi-

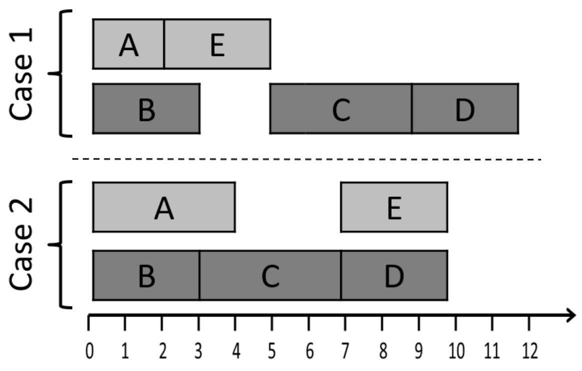


Figure 2.3: Timing anomaly example

ple successor states. Since modeling all possible transitions between abstract states rapidly becomes computationally intractable, STA approaches typically discard those states that are unlikely to lead to the global worst-case behavior. In particular, the underlying assumption in STA approaches is that timing can be safely analyzed at the level of single execution blocks as a function of an initial state and a given input, so that local worst-cases transitions are always assumed to lead to the global worst-case timing. Timing anomalies clearly spoil this assumption, forcing STA to take the appropriate countermeasures.

A relevant STA timing anomaly classification looks into the effects on timing on the analysis scope, distinguishing between bounded and unboundable timing anomalies [158]. Anomalies are classified as *k-bounded* if their effect can be factored in by adding a conservative (possibly overly-pessimistic) constant to the computed WCET bound. Instead, to account for unboundable timing anomalies, leading to the so-called *domino effect*, STA is forced to consider all possible states and transitions, which is evidently untenable.

In contrast with the extended research existing on timing anomalies for STA, there is still no definition or way to handle timing anomalies on MBTA or MBPTA.

2.1.6 Time composability

With the performance increase of CRTES, more and more tasks and functionalities are being integrated in the same system (for instance in a System on Chip (SoC)). However, integrating several tasks adds complexity to their design and integration.

The timing of these tasks needs to be assessed against their allocated time budgets early in the design process to take corrective actions with limited effort and time costs [112]. Otherwise, not only regression tests are more complex to design but also, as software gets integrated, finding an overrun late can cause costly system redesigns and even delay product's time to market. Hence, it is desirable for system engineers to have *time composable* timing bounds that are estimated in early design stages and remain valid upon integration of other software components, thus enabling *incremental software integration.* This is a fundamental property in increasingly-complex multi-provider software projects like IMA [1] in avionics or AUTOSAR [62] in automotive domains.

Typically, there exists a tradeoff between time composability and tightness. This happens, for instance, in systems where resources are shared amongst tasks. In such systems, a common way to make tasks time composable is to assume that contender tasks are always using the shared resource whenever possible, thus increasing allowances needed on the task under analysis but guaranteeing that the derived WCET estimates will hold no matter the usage of resources (activity) made by the contenders.

2.2 Caches in Real-Time Systems

Cache memories are one of the processor components with highest impact on performance. Most HPC processors include several cache levels: Intel Core family, AMD Ryzen, IBM Power 9 etc. Processors in the CRTES domain also include caches like the NXP T2080, Cobham Gaisler NGMP, and ARM Cortex A9/A53.

Caches have a key impact on the WCET estimates and present challenges to the CRTES industry. Mainly, they hinder obtaining sound WCET estimates, since they introduce jitter in the execution time. Without caches, memory accesses usually have longer yet constant access time; while with caches they can result in either a (shorter) hit, or a (longer) miss.

Despite the complexity caches add to timing analysis, the potential performance benefits of caches have resulted in several research works to enable their use in CRTES. In the context of STA, cache analysis has been used in order to predict the WCET by analyzing the cache behavior [36]. However, these techniques are usually limited to simple single-level cache systems, with few works tackling the challenge of multi-level caches in STA.

With MBPTA randomizing the timing behavior of the cache placement policy, the cache replacement policy, or both, representativity is easier to achieve. In particular, random placement relieves the end user from controlling the memory location of code and data – and the impact it has on timing – since addresses have no effect on placement anymore. Random Replacement (RR) instead removes systematic pathological performance cases, thus enabling tighter WCET estimates with MBPTA. In the next subsections the different placement and replacement policies currently existing for MBPTA are introduced.

2.2.1 Placement

The placement policy determines the cache set accessed by a given memory address. It is important that a single address always maps to the same set, or else it would not be retrievable. Figure 2.4 shows a block diagram of a cache and how randomized placement would fit in its overall design. As shown, for the generation of the index – used to feed standard modulo placement – specific logic combines the accessed address and a random number from a pseudo-random number generator (PRNG). State-of-the-art PRNGs deliver value series long enough to exclude repetition in short periods,

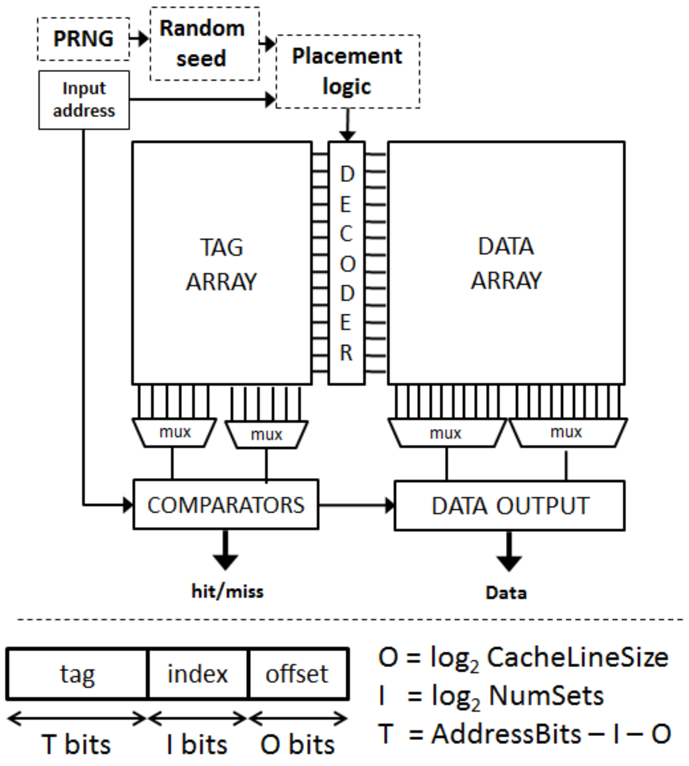


Figure 2.4: Hardware schematic of a random placement cache

thereby preventing any potential correlation of events [9].

2.2.1.1 hRP placement

The hash Random Placement (hRP) policy [92] uses a parametric hash function whose input includes the memory address to be accessed (factoring out those determining the offset within the cache line) and a random seed. It produces the (random) set where the address is placed with that random seed. The hash function uses a set of rotator blocks and XOR gates so that the set chosen for any given address is random. Thus, whether two addresses are placed or not in the same set is a random event. Upon change of the random seed, addresses are randomly and independently mapped into sets. hRP provides homogeneous distribution of addresses across sets, so the probability of each address to be placed in each set is $1/S$, where S is the number of sets.

hRP is used by flushing cache contents and setting a (new) random seed, usually at task execution boundaries[1]. This leads to a random placement of addresses, that holds during the whole execution, so addresses placed in the same set compete for the set

[1]Tasks sharing a cache memory require coordination for seed update and cache flushing. This can be achieved by changing seeds at execution time partition boundaries as described in the context of IMA and proven in [153].

space during the whole run, whereas addresses placed in different sets have no conflict in that run. hRP imposes that cache line alignment during analysis and operation is preserved. Thus, objects can be shifted in memory freely at the granularity of cache line size upon integration without impacting (random) placement.

2.2.1.2 Random Modulo (RM) placement

Unlike hRP, RM placement [72] breaks the dependence between memory location and cache placement while preserving the advantages of spatial locality as the standard modulo placement does. In particular, RM prevents conflicts between cache lines with identical tag bits, which we refer to as a cache segment. This is achieved by using a random seed, hashed with tag bits (T bits), that determines a random permutation of I (index) bits. Such random permutation changes across addresses by varying T bits and across random seeds. Thus, addresses are placed in random and independent sets across runs. However, two addresses with identical T bits and different I bits are necessarily placed in different cache sets given a fixed random seed. Thus, nearby addresses (those sharing the same T bits) cannot be placed in the same set.

RM poses constraints on integration: addresses in a cache segment (same T bits) during analysis must belong to the same cache segment upon integration. Hence, addresses may be shifted at the granularity of 2^{I+O} bytes (the size of a cache segment), which is practically achieved by making cache way size (W_{size}) match that granularity ($W_{size} = 2^{I+O}$), which is further made match memory page size. Thus, at software level objects are aligned at page boundaries and cache ways need to match that size (or be divisors of that size). As discussed later in this Thesis, this is a good tradeoff for L1 caches, but not for L2 caches.

2.2.2 Replacement

The replacement policy selects, in an associative cache, which line to evict from a set. The main cache replacement policy used in MBPTA systems is the RR.

2.2.2.1 Random replacement

RR evicts a specific cache line with a probability of $1/N$ for an N-way cache. Hence, the probability of a line surviving an eviction is $(N-1)/N$, and hence, there is a non-null probability of survival for cache lines for any access sequence. In particular, even if the cache set space is exceeded (e.g. placing 5 lines in the same set in a 4-way set associative cache), RR provides a survival (hit) probability larger than 0 by construction (e.g. $\left(\frac{3}{4}\right)^4 = 0.316$ for a cache line after 4 replacements in a 4-way cache).

On the other hand, if we consider a sequence with N addresses instead of $N+1$, then we realize that the most popular deterministic policies (e.g. Least Recently Used (LRU), FIFO) would lead to all-hit sequences except for cold misses. Instead, RR has non-null probability of evicting some cache lines before (randomly) placing all addresses in distinct physical cache lines so that all remaining accesses become hits.

This occurs because RR is unaware of temporal locality since it does not preserve any history.

Since pWCET estimates with MBPTA need to account for the worst case that can occur with non-negligible probability, pathological cases occurring with relevant probabilities need to be accounted for. In the case of RR, some low-probability high-execution time eviction scenarios need to be accounted for, but those scenarios will not include the absolute worst case, which is the actual case for deterministic policies under systematic pathological cases. Still, the fact that these eviction sequences can occur under RR may lead to high pWCET estimates, which relates to the fact that RR is a locality-unaware replacement policy.

2.2.3 Write policy in multilevel caches

When designing a multilevel cache hierarchy there are several design choices to be made, which are not independent of each other but quite tightly correlated. The write policy is an important design choice that can have a big impact in the resulting design [69]. Mainly two policies exist: write-though (WT) and write-back (WB). With WT, every time a write to the cache is made, the write propagates to the upper level. On the contrary, with WB when the write is made the line is marked as dirty, and the write to the upper level will only be made when the dirty line is evicted.

Another decision with writes to caches is whether to fetch a cache line on a write miss, i.e. write allocate (WA) or no-write allocate (nWA) [69]. With WA, on a write miss data is fetched into cache, as it is the case for read misses, and once fetched, the write operation occurs. With nWA, on a write miss the write operation is simply forwarded to the next cache level (or memory). Both WT and WB can use either of these write-allocation policies, but we only consider WB-WA and WT-nWA caches in this Thesis, since they are the most common choices. However, our research work presented later could be extended to other combinations.

2.2.4 Cache inclusion

The *inclusivity* of the lower cache levels into the upper cache levels (those closer to memory), imposes that all contents in the lower level cache are also included in the upper level cache [69]. Hence, whenever a cache line is evicted from the upper level cache, all cache lines in the lower level cache holding any of the contents of the cache line evicted in the upper level cache are also evicted. Under an *exclusivity* approach, cache lines can be stored only in one of the two levels involved. When a new cache line is fetched by the processor, it is typically fetched into the lower level and removed from the upper level. When a cache line is evicted from the lower level it is moved up to the next level. Finally, *non-inclusive* caches are those where no constraint is imposed on whether cache lines are stored in upper or lower cache levels. This is a common choice for instruction caches since they are typically read-only and thus, cache lines can be simply removed on an eviction.

2.2.5 Reliability

In the CRTES industry, one of the main requirements of the systems deployed is a certain level of reliability. This means that the system must operate correctly and in a timely manner under a variety of conditions, which will depend on the specific field and the criticality of the task. A potential system disruption are soft errors. Soft errors can be caused, for instance, by atmospheric neutrons [116] The result can be a bit flipped from 0 to 1 or from 1 to 0. If this bit is accessed by the program, it can result in an error.

This can happen with a greater chance in cache memories due to their relative size with respect to the core. In order to detect and fix soft errors in caches, some sort of redundancy mechanism is added to the cells. Error Correction Codes (ECC) [66] are a common solution. ECC add extra bit cells per cache line to check if there is an error and correct it. The number of bits per line that can be detected to be flipped and corrected in case of a flip depend on the number of extra bits added and the detection and correction algorithms. A simple solution is Single Error Correction, Double Error Detection (SECDED), which can detect two errors and correct one.

Chapter 3

Experimental Setup

3.1 Methodology

In the context of this Thesis, the methodology used in order to evaluate new ideas or approaches is as follows, see Figure 3.1:

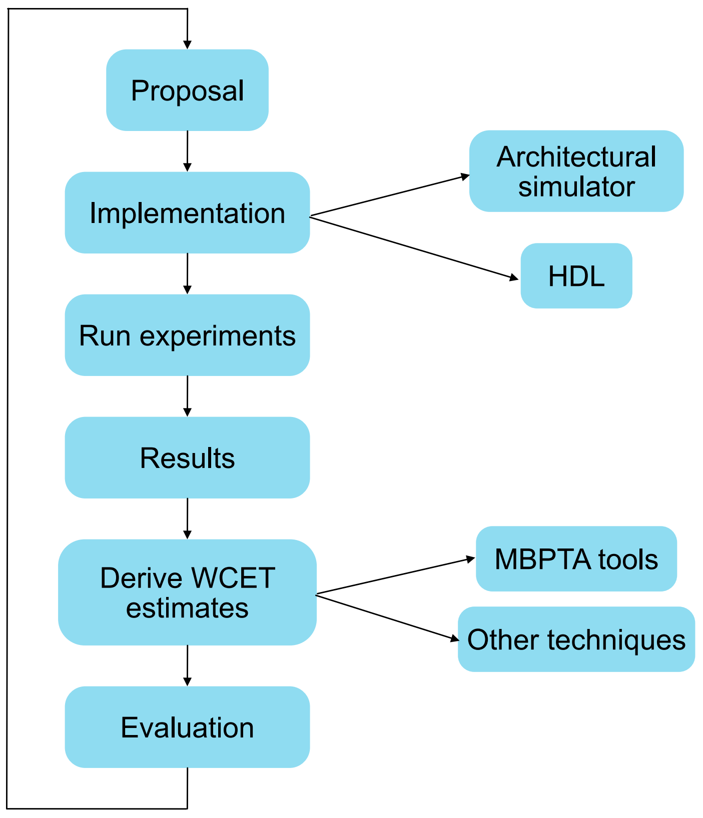


Figure 3.1: Methodology used in the evaluation of the Thesis

1. **Proposal**. A proposal, i.e. change in the design of the processor, is made to capture specific challenges.

2. **Implementation**. The proposal is implemented in the architectural simulator or in the Hardware Description Language (HDL) implementation in an Field-Programmable Gate Array (FPGA).
3. **Run experiments**. Experiments are run with an appropriate set of benchmarks on a representative hardware setup.
4. **Results**. Statistics from the exection in either the architectural simulator or the FPGA are collected. Some statistics that could be interesting are for example execution cycles, cache misses, bus requests etc. The set of specific statistics depends on what the expected effect of the proposal is.
5. **Derive Worst-Case Execution Time (WCET) estimates** In order to derive WCET estimates a process must be followed depending on the particular timing analysis used. In our work, we mainly use Measurement-Based Probabilistic Timing Analysis (MBPTA), so we use a specific set of tools to derive Probabilistic Worst-Case Execution Time (pWCET) estimates.
6. **Evaluation**. After we have the results and the WCET estimates, we can compare results with our target. If they don't reach our target, we can propose further changes and start again the process.

3.2 Reference processor architecture

The reference processor we are going to use is a multicore, multi-level cache representative of the recent trend in the processors used in critical real-time domains. In particular, we model Cobham Gaisler's Next Generation Microprocessor (NGMP). This chip is a solid candidate for future missions of the European Space Agency (ESA). Competing companies have chips with similar configurations with few simple cores (2-8) and 2 level cache hierarchy (ARM Cortex R5 [16], ARM Cortex M7 [15], Freescale PowerQUICC [136], Freescale P4080 [119], etc).

To better understand the particularities of the specific platform used, we provide some background on the LEON core family. Afterwards, we explain more in detal the NGMP SoC used.

3.2.1 LEON 3

The LEON 3 [60] processor core is compliant with the SPARC V8 architecture. It features an advanced 7-stage pipeline that can be clocked up to 125MHz in an FPGA and 400MHz on 0.13 μm ASIC technologies. The pipeline stages are: fetch (F), decode (D), register access (RA), execution of non-memory operations (Exe), dL1 access (M), Exceptions (Exc) and write back (WB). The execution units are equipped with an integer and a floating-point unit (FPU).

Designs of SoC based on the LEON 3, such as the GR712RC, have private L1 instruction and data caches. The GR712RC also features two cores without any shared cache. One particularity of this SoC is that the designer, Cobham Gaisler, provides an IP library (called GRLIB) under GNU GPL license. GRLIB provides the

LEON 3 core as well as several buses such as the AMBA bus or the SRAM controller, amongst others.

3.2.2 LEON 3+

The LEON 3 processor has been extended as part of the PROXIMA project [128] to support a probabilistic timing analysis platform. We will call this design the LEON 3+ [70]. The main changes done to the platform were in order to reduce jitter. This has been achieved by both:

1. Making the floating point operation latencies constant to the maximum.
2. Changing cache placement and replacement policies to introduce randomization in the timing behavior.

These are the cores that we will use in our work, since they set the baseline design that enables probabilistic timing analysis.

3.2.3 LEON 4

The LEON 4 [58] core is based on its predecessor the LEON 3. It is also a SPARC V8 processor with a 7-stage pipeline, although this one includes branch prediction. LEON 4 runs at 150MHz in an FPGA and at 1.500MHz in a 32nm ASIC.

Designs based on the LEON 4 such as the GR740 now have 4 cores and introduce an L2 cache shared amongst all the cores. They also introduce the changes made in the LEON 3+ to allow for probabilistic timing analysis.

The designer also specifies that the LEON 4 SoCs are fully parametrizable with HDL generics, so several custom-tailored SoCs can be made.

3.2.4 NGMP

In particular, the SoC that we will mainly use in this Thesis, is the Next Generation MicroProcessor (NGMP) [59]. This processor has been designed with future spacial missions in mind, specifically for the European Space Agency (ESA). This design features four LEON 4 cores, each with its own private L1 instruction and data caches, connected by a bus to the shared L2 cache.

The only change we make to the standard NGMP is that our design uses LEON 3+ cores. This is due to the public unavailability of the LEON 4 core HDL. However, although the core is different, it has the same characteristics that enable us to use probabilistic timing analysis. Furthermore, for the rest of the changes in the SoC, mainly the new L2 shared cache, is still the standard NGMP one.

We model 4KB 4-way dL1 and iL1 caches per core, with 16 and 32B/line respectively, and a shared 512KB 4-way L2 cache that can be partitioned so that each core receives an independent L2 cache way of 128KB. The dL1 is write-through no write-allocate, so store operations are always forwarded to the L2 cache and do not fetch data to dL1 on a miss. The L2 is write-back write-allocate, so on a store miss, the

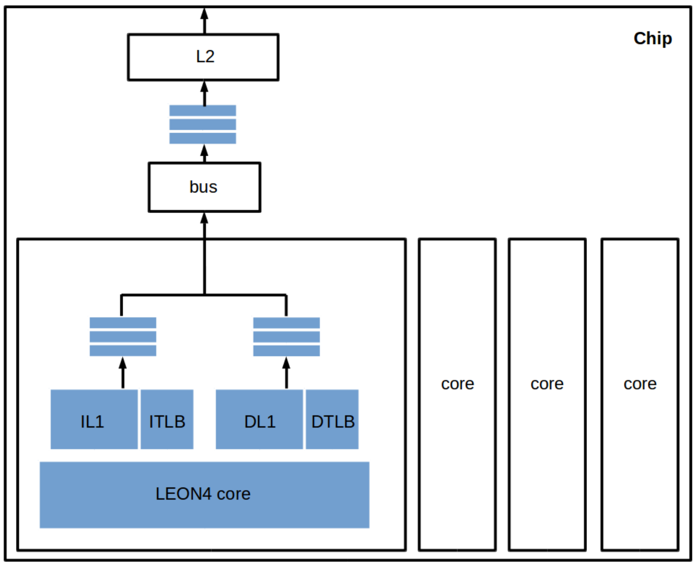


Figure 3.2: Architecture schematic of the NGMP SoC

cache line is fetched into L2 – and modified. Caches are non-inclusive, so no control is exercised on whether cache lines must or must not reside in any particular cache memory. Bus arbitration implements Random Permutations (RP) [82]. RP are also used to arbitrate memory requests (L2 cache misses). Memory latency is 16 cycles to serve a request and 27 cycles until the next request is started [125].

3.3 Architectural simulator

The architectural simulator used in this Thesis is SoCLib [138], which is Cycle Accurate/Bit Accurate (CABA). Specifically, we use a modification of SoCLib that models the NGMP SoC. The version of the SoCLib simulator that has been used in this Thesis has several modifications to tune it to the NGMP specific architecture, with results in less than 3% error on average [80].

The simulator uses the instructions generated by an emulator that can emulate both the SPARC and PowerPC ISAs. As part of this Thesis we adapted the simulator to also work with instruction traces that were generated running the benchmarks on QEMU for SPARC. We also modified several emulator instructions for SPARC to provide more information to the simulator. These modifications have enabled the work in Chapter 7 about redundancy, since specific dependencies between registers were needed also on the simulator.

Several changes have been also introduced in the simulator, basically to support new features. New placement (Chapter 4) and replacement (Chapter 5) policies have been implemented, as well as a hybrid write policy (Chapter 6). We used the SoCLib simulator in all the contributions of the Thesis, except the contribution in Chapter 9 that has exclusively been done with the FPGA implementation (explained in the following subsection).

3.4 Energy simulator

In addition to the performance improvements, in some contributions the energy consumption of the solution has been taken into consideration. We used CACTI [115] to obtain energy results. CACTI is a cache and memory simulator that computes access time, cycle time, area, leakage and dynamic power. It is one of the most used energy simulators in the research community.

The CACTI simulator has been modified to take into consideration different ECC solutions, so that the overhead of the additional memory cells could be considered. For the rest of the experiments the latest CACTI version (6.5) has been used.

3.5 FPGA implementation

In order to implement and test some of our proposed techniques, we use an FPGA board with the HDL implementation of the NGMP. Cobham Gaisler has provided the basline implementation of this board, since it has collaborated with our research group in the PROXIMA FP7 European project [128]. In order to generate the bitstreams needed for the FPGA, we use the programmable logic device design software Altera Quartus II [11]. The FPGA board used is a Terasic Stratix-IV able to operate at 100Mhz.

As explained in the previous section, the NGMP has some random placement and replacement functions already implemented to allow for probabilistic timing analysis. The main issue we have with the implementation is that all the code is not available to us. Some parts, such as the L2 cache, are closed source, so we could not implement all the techniques in the FPGA. For the techniques that could not be implemented, we used the architectural simulator.

All in all, the FPGA implementation of the NGMP has been used in 2 contributions of this Thesis:

- Placement policies. We implemented a new placement policy. More about this implementation is found in Chapter 4.
- Timing anomalies. We obtained the timing samples from the FPGA board, see Chapter (Chapter 9).

3.6 MBPTA tools

As explained in section 2.1.4, to apply MBPTA two elements are required: to handle the sources of jitter to ensure representativity, and to use statistical analysis techniques to compute the pWCET curve. The sources of jitter are handled by the LEON 3+ processor, either by upper bounding or randomizing the sources of timing variation. In this section we describe how to handle the second part: applying EVT to obtain pWCET curves.

In order to obtain pWCET curves we use the methodology presented in [5]. The basic idea is that obtaining enough variability in the analysis runs can allow us to derive an exponential curve that will upper-bound any possible timing in operation. However, for this process to be valid, the execution times obtained must meet some criteria. Let us go through this process to explain the different challenges. In Figure 3.3 we see the overall process for deriving a pWCET curve from the analysis runs.

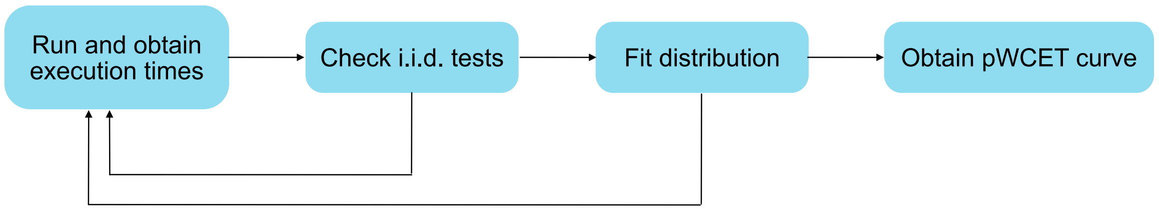


Figure 3.3: Schematic of the procedure to derive a pWCET

The first step is to obtain different execution times from running the program a number of times. The variability in the results could be, for instance, due to the timing randomization of some components.

Once these timings have been collected, independence and identical distribution (i.i.d.) tests have to be passed (step 2 in the figure). Specifically we use the Ljung-Box independence test [34] and the Kolmogorov-Smirnov two-sample i.d. test [53].

Empirically we have found that with around 1000 execution runs, we can usually obtain representativity that passes the iid tests, although it can depend. For example in some cases with just a few hundred runs the tests can already be passed.

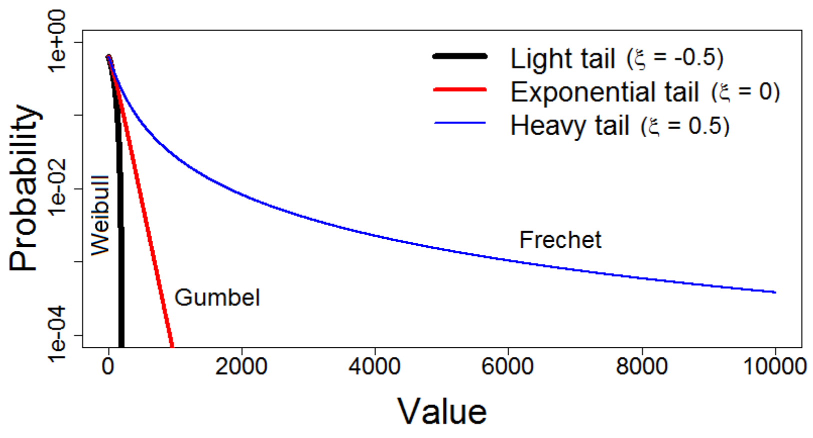


Figure 3.4: Types of tails the pWCET curve can have (from [5])

Then, the distribution is tested to see what kind of tail it has. In order to derive the pWCET, the distribution should be upper-bounded by an exponential tail, as shown in Figure 3.4. If it is a heavy tail, instead, more runs need to be collected. For fully understanding the reasoning behind the types of tails, we direct the reader to the full work [5].

Once the tests are passed, the sample points are fitted to an exponential distribution (step 4 in the figure), and a pWCET curve that upperbounds the tail of the distribution under analysis is derived.

Table 3.1: EEMBC Automotive benchmarks

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

3.7 Applications

In this Thesis we use different benchmarks and applications, mainly related with the real-time systems industry. The main benchmark suite we use is the EEMBC Automotive [127]. As additional benchmarks we sometimes use Mediabench [101], Mälardalen [64] and a railway case study. In this section, we first explain the different benchmark suites, and finally we explain the limitations and use cases of each of them and how are they used in this Thesis.

3.7.1 Benchmarks and case study

- **EEMBC Automotive**. The EEMBC automotive suite [127] is developed by the Embedded Microprocessor Benchmark Consortium with the objective of measuring the performance of Embeded Systems with the use of programs used commonly in automotive systems. The structure of these benchmarks is the following: each program has a main loop, executed a number of iterations (configurable by the user) with some calls in the loop body. The number of iterations we use is the default one provided with each benchmark. The input data is already included in the benchmark suite. In Table 3.1 we can see the list of benchmarks of this suite.
- **Mediabench**. The Mediabench benchmark suite [101] is open source and comprises a set of media algorithms, for instance for video, audio, image, security... Most benchmarks come with a separate decode and encode variant. We use the default input files that come along with the suite. In Table 3.2 there is a list of the benchmarks and what their task is.

Table 3.2: Mediabench benchmarks

Name	Description
adpcm.d	Adaptive differential pulse-code modulation, decode
adpcm.e	Adaptive differential pulse-code modulation, encode
epic.d	An experimental image compression utility, decode
epic.e	An experimental image compression utility, encode
g721.e	Voice compression G721, encode
g721.d	Voice compression G721, decode
gsm.d	GSM 06.10 standard for full-rate speech transcoding, decode
gsm.e	GSM 06.10 standard for full-rate speech transcoding, encode
jpeg.d	Image lossy compression JPEG, decode
jpeg.e	Image lossy compression JPEG, encode
mesa.m	3-D graphics library clone of OpenGL, mipmap application
mesa.o	3-D graphics library clone of OpenGL, osdemo application
mesa.t	3-D graphics library clone of OpenGL, texgen application
mpeg2.d	Standard for high-quality digital video transmission, decode
pegwit.d	A program for public key encryption and authentication, decode
pegwit.e	A program for public key encryption and authentication, encode
pgp.d	Security algorithm PGP, decode
pgp.e	Security algorithm PGP, encode
rasta	A program for speech recognition

- **Mälardalen**. The Mälardalen benchmark suite [64] consists of various open source programs. Each application has its own specific input set, and the structure is fairly simple with mostly linear code and a few loops. Table 3.3 shows the programs used.
- **Railway case study**. As a real world case study, we use the safety function part of the European Train Control System (ETCS), called European Vital Computer (EVC). This study [110] consists of 10 different parameter configurations, each using different combinations of speed and acceleration. Each configuration is used as a different case, labelled from 0 to 9 in our results.

3.7.2 Limitations and use cases

The 3 different benchmark suites and the case study have not all been used in each contribution. Not all benchmarks were able to be executed in the FPGA board, since the executions were on bare metal and some benchmarks relied on operating system calls.

The most representative benchmarks for the domain we focus in are the EEMBC Automotive. However, the current trend to bigger, more complex software is not always reflected in the EEMBC Automotive benchmarks. Specifically, we found that most programs could fit in the L2 cache of a processor such as our reference one, the Gaisler NGMP. In order to make a more complete study, we complemented the

Table 3.3: Mälardalen benchmarks

Name	**Description**
bs	Binary search for the array of 15 integer elements
bsort	Bubblesort program
cnt	Counts non-negative numbers in a matrix
compress	Data compression program
cover	Program for testing many paths
crc	Cyclic redundancy check computation on 40 bytes of data
duff	Using "Duff's device" from the Jargon file to copy 43 byte array
fac	Calculates the faculty function
fdct	Fast Discrete Cosine Transform
insertsort	Insertion sort on a reversed array of size 10
jfdctint	Discrete-cosine transformation on a 8x8 pixel block
qsort-exam	Non-recursive version of quick sort algorithm
qurt	Root computation of quadratic equations
select	Selection of the Nth largest number in a floating point array
statemate	STAtechart Real-time-Code generated code
ud	Calculation of matrixes

EEMBC Automotive with Mediabench that for some benchmarks does not fit in L2.

Furthermore, in some specific scenarios we also used the Mälardalen and the Railway case study to provide a more comprehensive evaluation when the EEMBC Automotive and Mediabench were not representative enough. For example, in the Chapter were we experiment with replacement policies (Chapter 5), we could not see significant impact on EEMBC Automotive or Mediabench, since the nature of these benchmarks meant that no significant reuse was being done in L1 cache levels, and thus the replacement policy had little impact. In this case we used the Mälardalen and the Railway case study.

Chapter 4

Cache placement policies

4.1 Introduction

Time randomized caches simplify deriving trustworthy Probabilistic Worst-Case Execution Time (pWCET) estimates that upper-bound deployment-time execution times. In this line, as introduced in section 2.2, mainly two random placement policies have been proposed: hash-based random placement [92] and Random Modulo (RM) placement [93]. As previously stated in Sections 2.2.1 and 2.2.2, modern real-time systems usually have several levels of cache. For instance the ARM Cortex R5 [16], ARM Cortex M7 [15], Freescale PowerQUICC [136], Freescale P4080 [119] all have at least 2 levels of cache. Despite the existence of several cache placement proposals with random policies that enable Measurement-Based Probabilistic Timing Analysis (MBPTA), it is not yet well understood how to design efficient multi-level time-randomized cache hierarchies and how different randomization policies in each level impact average performance and Worst-Case Execution Time (WCET).

In order to fill this gap, our contributions in this Chapter are as follows:

- We perform a design space exploration of multi-level random cache designs in a cycle-accurate simulator. We explore monolithic designs by applying existing L1 placement policies to both L1 and L2. We show that these policies are not designed for L2 caches and have performance (average/worst-case) or time composability issues.
- To tackle the observed deficiencies, we introduce, for the first time, hierarchical placement designs that solve L2 related issues while still being MBPTA compliant. Our results show that the proposed hierarchical designs have no negative impact on average performance, improve worst-case results with respect to the monolithic designs, and favor time composability.
- We implement and integrate the most cost-effective cache hierarchy in our NGMP RTL model. Our results show that it has almost the same performance as modulo placement, provides tight WCET estimates, and enables MBPTA time-analyzability.

4.1.1 Time composability in caches

As introduced in the background (Section 2.1.6), time composability allows system engineers to have timing bounds that remain valid from early design stages until integration with other software components, enabling incremental software integration and reducing the costs and time of the project.

With caches, the relative position of program's memory objects may change across software integrations leading to different cache layouts with arbitrary impact on execution time. This breaks time composability and shifts timing analysis and verification to the latest design stages (when the binary is fixed) with increased risk of failing to meet execution time bounds. In this context, random placement policies together with MBPTA have been shown to enable incremental verification in the presence of cache memories [38]. In particular, random placement policies break the dependence of cache placement on the actual memory addresses, i.e. in each run software experiences random placement of memory objects in cache. As a result, the actual memory addresses are irrelevant for cache placement and the space of potential cache placements is randomly sampled in each run. Since the probability distribution for cache placements observed at analysis matches that during operation, impact of cache placement can be analyzed with MBPTA to produce probabilistic bounds on its impact on execution time. In fact, MBPTA is capable of considering different sources of random variation (e.g. cache placement for multiple caches, random arbitration in buses) simultaneously. However, while random caches remove WCET estimate dependence on memory location of objects, thus relieving the user from controlling memory placement, it has not been explored how the different random placement policies need to be combined into multi-level cache hierarchies. In particular the desired properties are:

1. **WCET reduction** as the main metric to optimize.
2. **Reduced impact on average performance** due to the importance of this metric for mixed-critical scenarios executing tasks with different criticality levels.
3. **Increase time composability** to favor incremental software development as described above.
4. **MBPTA compliance** to reduce the cost of changing existing timing analysis tools.

Next, we present several multi-level cache designs and assess them against these metrics. Multi-level cache designs build upon the two main random placement policies that enable MBPTA, namely hash Random Placement (hRP) and RM, already introduced in Section 2.2.

4.2 Multi-level Random Cache Approaches

Time-randomized cache placement policies have been evaluated mainly for single-level cache hierarchies. An exception to this is hRP that has been shown to keep

Table 4.1: Placement policies analysed

Setup	**MOD**	**hRP**	**RM**	**hRP2**	**hRP+MOD**	**hRP+RM**
L1	MOD	hRP	RM	RM	RM	RM
L2	MOD	hRP	RM	hRP	hRP *PL* + MOD	hRP *PL* + RM

its MBPTA-compliance properties for multi-level caches [93]. However, hRP is the existing random placement policy with lowest average and guaranteed performance. Hence, there is significant room for improvement in multi-level random cache design. In this line, this section presents several approaches that use, individually or in a smartly-combined way, different random placement policies to provide higher-performance MBPTA-compliant multi-level cache designs. For clarity, we use the L2 placement policy as the identifier for the multi-level configuration. See Table 4.1 for the list of configurations.

4.2.1 L2 Monolithic Placement

4.2.1.1 MOD setup

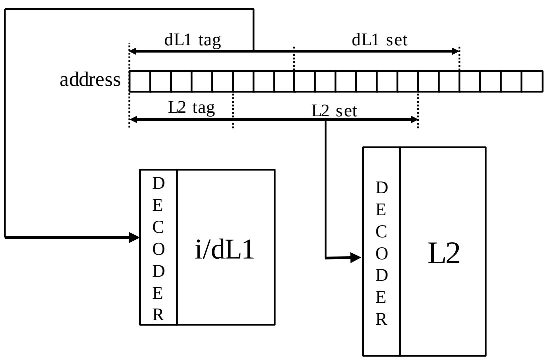
MOD setup is the reference setup against which we compare other randomized setups in terms of average performance. This setup uses MOD placement – deployed in many multi-level cache designs as placement policy for all cache levels – see Figure 4.1(a). It determines cache placement based on cache index bits (I bits) and it is not amenable for MBPTA. This is mainly due to MOD deterministic behavior: although conflictive memory alignments can be infrequent, they may occur upon integration with a systematic and pathological nature, resulting in the so-feared (for measurement-based techniques) cache risk patterns.

4.2.1.2 hRP setup

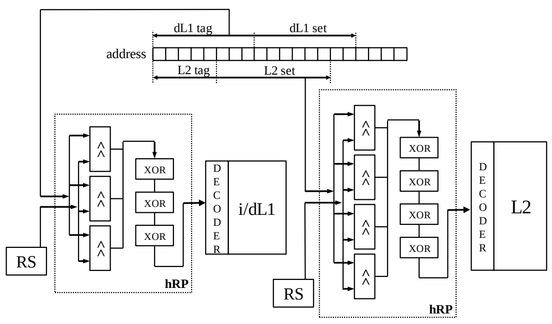
In this setup hRP placement is used for first level data and instruction caches, respectively referred to as dL1 and iL1, and the second level cache (L2), see Figure 4.1(b). This setup was already considered in [93] given that hRP was the first random placement policy proposed compatible with MBPTA. This design only imposes preserving cache line alignment between analysis and operation phases. However, hRP allows all cache lines to be randomly placed completely independently. Therefore, few cache lines may be placed in the same cache set in L1 caches (either dL1 or iL1) with a relevant probability for pWCET estimation, and also in L2 cache. Thus, while those bad placements occur with relatively low probability, having low impact on average performance, they may lead to large impact in pWCET estimates to account for very bad placements that can occur even with very small working sets.

4.2.1.3 RM setup

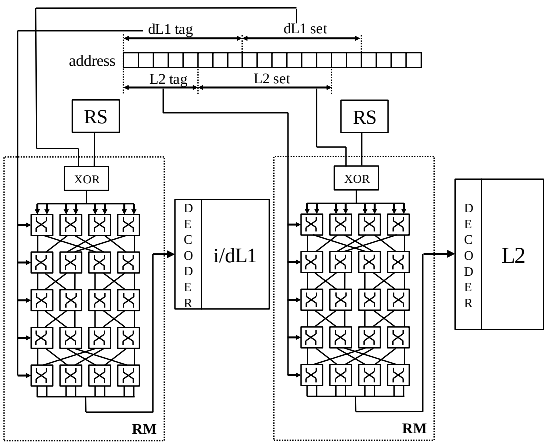
RM placement implements a Benes network (Figure 4.1(c)) that produces a random permutation of the index bits – XORed with the random seed – being the permutation



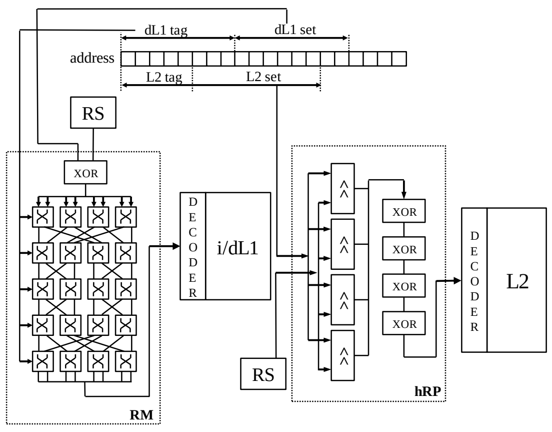
(a) Baseline Modulo (MOD) two-level cache



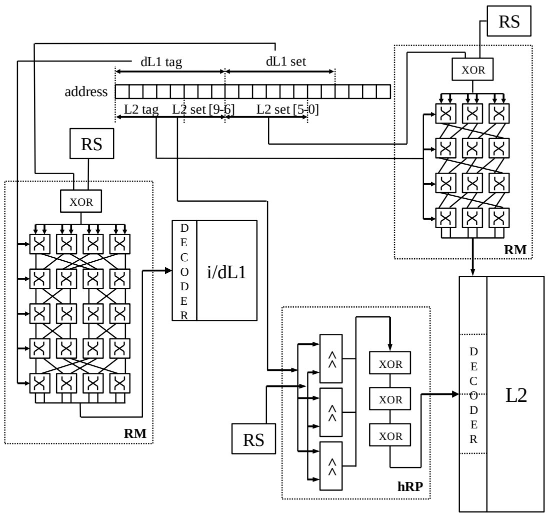
(b) hRP in both levels



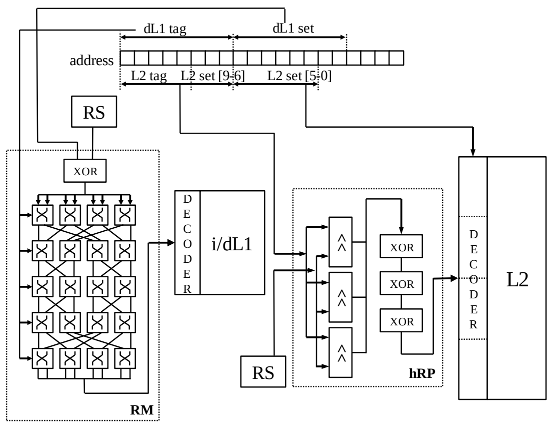
(c) RM in both levels



(d) RM in dL1/iL1 and hRP in L2



(e) RM in dl1/iL1 and hRP+RM in L2



(f) RM in dl1/iL1 and hRP+MOD in L2

Figure 4.1: Hardware schematic of the different placement policies implemented and analysed

controlled by the T tag bits. With *RM*, cache-segment alignment must be maintained between analysis and operation: all addresses fitting in a cache segment in the experiments carried out at analysis, must remain in a segment during operation. As explained before, the Real-Time Operating System (RTOS) can easily achieve this goal by matching memory page size with cache segment size, or making page size be a multiplier of cache segment size. In fact, this assumption has already been shown compatible with complex avionics case studies [153].

RM can be soundly used for first level caches whose way size (i.e. cache segment size) is typically equal or smaller than the page size. When the way size is larger than the page size, usually the case for L2 whose size is easily above 128KB-256KB, then *RM* can be used if the RTOS preserves page alignment at that granularity. For instance, if the cache way size is k times larger than the page size, the RTOS should maintain the alignment of pages at $k \times page\_size$ bytes granularity to soundly apply MBPTA. However, dealing with this constraint is unaffordable in practice due to memory fragmentation (the subsequent memory space waste). Further, it also adds complexity to the RTOS. Hence, *RM* can be used in L1 caches, and *hRP* in the L2 instead as presented next.

4.2.1.4 Summary

Neither MOD nor RM in L2 are MBPTA compliant and defeat achieving time composability. We keep the former for average performance comparison purposes, while we discard the latter. hRP is MBPTA compliant and maintains time composability, and hence, we use it as reference randomization policy for L1 and L2.

4.2.2 L2 Hierarchical Placement

Next we propose hierarchical designs based on multiple policies to get the best of each policy.

4.2.2.1 hRP2 setup

This setup combines the advantages of RM in L1 caches to preserve spatial locality, and the flexibility of hRP in L2, avoiding posing undue constraints on memory object alignment, see Figure 4.1(d). Thus, qualitatively, this setup is far more convenient than those presented so far by smartly combining in different cache levels appropriate random placement policies. However, hRP has been shown to have low but non-negligible hardware cost, due to the expensive barrel shifters followed by a tree of XOR gates, whose number and size – and so impact on area – grows significantly with the number of address bits to handle. Therefore, we examine other hybrid solutions for L2 caches.

4.2.2.2 hRP+RM setup

This setup – that uses RM in L1 caches – performs hRP at page size in L2 and RM within page size, as seen in Figure 4.1(e). We build on the observation that, as L2 cache ways are conceptually split into cache segments, hRP can be used to randomly select the cache segment where an address is placed and RM to select the set within the segment. This setup requires preserving page alignment between analysis and operation phases. However, such constraint is already imposed by L1 caches, so constraints remain the same as for any other setup using RM in L1 caches.

This hierarchical design has a positive impact in the implementation cost of the L2. First, hRP only operates on tag (T) bits instead of on $T + I$ bits. RM, instead, randomizes placement within page boundaries thus operating on the remaining I bits. However, RM is much cheaper than hRP in terms of area. This is further detailed later in Section 4.3.2 and in Table 4.3. While the impact on the critical path is roughly null, hRP logic becomes the critical path for large caches (larger than L2 caches evaluated in this work). Hence, the hybrid solution would also mitigate delay issues in those cases.

As this design uses RM at the page level, the low performance of hRP is mitigated. The other side of the coin is that it can be the case that two pages are randomly mapped to the same cache segment. The fact that we use RM inside L2 segments reduces the likelihood that pages (segments) evict each other's lines systematically.

4.2.2.3 hRP+MOD setup

While the previous setup reduces the hardware overhead of L2 compared to those with hRP in L2, the hardware for L2 cache placement must still accommodate hRP across cache segments and RM within segments. This overhead can be further reduced by removing RM from L2 cache segments (and sticking to MOD), see Figure 4.1(f).

This approach reduces hardware complexity, but the degree of randomization achieved in L2 also decreases. While it has been shown that higher degrees of randomization lead to less abrupt execution time variations (and thus to lower pWCET estimates) [151], the fact that addresses go through RM placement and Random Replacement (RR) in L1 caches, hRP across cache segments in L2, and RR in L2 (if more than 1 way is used per core), already brings high degrees of randomization. Hence, we regard this setup as a good trade-off between randomization achieved and hardware cost. In particular, this setup decreases transistor count slightly and may reduce cache placement latency for caches with large number of L2 cache sets due to the decreased logic depth.

4.3 Evaluation

Our evaluation of this work is twofold. First, we make an extensive exploratory simulation with all the different placement approaches described. Later, we choose the best setup (hRP+MOD) and we implement it in RTL to compare it with the default MOD policy.

4.3.1 Simulator evaluation

We perform the evaluation in the architectural simulator described in Section 3.3. The configuration of the caches if the one described in subsection 3.2.4. We evaluate a large subset of both EEMBC Automotive and Mediabench, previously described in Section 3.7.

4.3.1.1 Results

Our designs aim at (a) maintain MBPTA compliance; (b) reduce pWCET estimates w.r.t. simpler MBPTA-compliant designs; (c) obtain comparable performance to non-randomized (and hence non MBPTA-compliant) modulo+Least Recently Used (LRU) based multilevel cache hierarchies; and (d) preserve time composability.

To assess MBPTA compliance, we have followed the approach proposed in [93] checking that cache events preserve its random/probabilistic nature. Our results – not shown for space constraints – show that an address can be randomly mapped to any dL1 (iL1) and L2 set. Further, independence and identical distribution tests on execution times are passed [5].

To derive pWCET estimates, and obtain solid average performance results, we carry out 500 runs for each benchmark-setup pair. pWCET estimates are shown for an exceedance threshold of 10^{-12} per run, since they are enough for the highest criticality software [153].

4.3.1.2 MediaBench

In Table 4.2 columns 2-3 show the pWCET estimates obtained with each placement policy normalized to the monolithic setup *hRP*. We observe that hierarchical setups consistently reduce the pWCET estimates of *hRP*, by 28% and 34% for hRP+MOD and hRP+RM respectively.

In terms of average performance, columns 4-6 show that the three hierarchical setups obtain comparable results to those of the deterministic approach (MOD+LRU), only up to 2% worse. This is so because bad placements occur seldom even for the worst setups, so average results hide outliers. We repeated the same analysis for executions resulting in the *highest 5% miss counts*, as they shape the tail of the execution time distribution, and hence WCET [5]. Our results show that hRP achieves worst results than hierarchical approaches: with hRP, by allowing each cache line to be placed randomly and independently in L2, any pair of cache lines can, in the worst cases, be placed in the same set and produce high miss counts, and hence execution times. This explains why hierarchical placements improve hRP for pWCET (columns 2-3).

4.3.1.3 EEMBC Automotive

EEMBC Automotive trends are similar to MediaBench. We observed similar average execution time for all placements, and hRP being the worst policy in terms of worst-case execution time. Results are omitted since they bring no further insights.

Table 4.2: Placement policies: MediaBench results

	pWCET vs hRP		avg perf vs MOD		
	hRP + MOD	hRP + RM	hRP	hRP + MOD	hRP + RM
ad.d	0.08	0.05	1.01	1.00	1.00
ad.e	0.25	0.18	1.01	1.01	1.00
ep.d	0.89	0.90	0.99	1.00	0.99
ep.e	0.90	0.65	1.00	1.00	1.00
gs.d	0.94	1.00	1.00	1.00	1.00
gs.e	0.75	0.75	1.01	1.01	1.01
jp.d	0.84	0.77	1.02	1.02	1.02
jp.e	0.91	1.10	1.01	1.02	1.01
m.m	0.99	0.89	1.01	1.01	1.02
m.o	0.48	0.49	1.00	1.00	1.00
m.t	0.87	0.83	1.01	1.01	1.01
pe.d	0.71	0.72	1.01	1.00	1.00
pe.e	0.65	0.64	1.01	1.00	1.00
pg.d	0.73	0.75	1.01	1.00	1.00
pg.e	0.69	0.16	1.02	1.00	1.01
rast	0.87	0.76	1.00	1.00	1.00

4.3.1.4 Summary

hRP+RM and hRP+MOD avoid some systematic effects of hRP, which reduces their L2 miss rate (and execution time) for pathological cases w.r.t. hRP. Also, it cannot be claimed whether hRP+RM or hRP+MOD is superior, since our results show that conclusions change across different benchmarks. Moreover, although the cost of implementing hRP+RM is only slightly higher than that of implementing hRP+MOD in L2, hRP+MOD can be regarded as an effective setup. Furthermore, this combination is interesting because it synergistically combines 3 different placement policies: RM in L1 caches, hRP across L2 cache segments, and modulo inside L2 cache segments. For these reasons, we implemented this setup in RTL.

4.3.2 Design Validation: RTL implementation

In order to validate the simulation results we implemented the hRP+MOD in the FPGA setup described in Section 3.5, with the parameters described in subsection 3.2.4.

4.3.2.1 Area overhead

Table 4.3 shows the area and delay overhead introduced by the different random placement functions considered in this study. As shown, RM requires significantly lower area than hRP since it uses a permutation network consisting of few pass transistors per index bit. The actual permutation carried out is driven by XORing

Table 4.3: Placement policies: hardware cost and delay

	dL1		L2		
	hRP	RM	hRP	hRP+MOD	hRP+RM
Trans. Count	49488	240	49440	24360	24600
Delay (ns)	0.65	0.26	0.52	0.52	0.52

address bits and seed random bits. Instead, hRP requires combining the seed random bits and the address by means of barrel shifters and several levels of XOR gates [92]. The hierarchical random placement implementation reduces area overheads of hRP by roughly 50%. This, coupled with the good performance results it provides, confirms the hierarchical approach as the most suitable option to implement random placement in L2 caches.

In terms of overall hardware occupancy, the baseline design occupied 70% of the resources in the FPGA, whereas the design including the random placement in all L1 caches and L2 occupies less than 72%, thus showing that all cache modifications required to achieve MBPTA-compliance incur very low overheads.

4.3.2.2 Critical Path

RM is faster than hRP-based approaches since the latency of the latter is mainly determined by the depth of the XOR gates tree employed to combine address and random bits. In the case of RM, the XOR gates tree must produce the bits required for configuring the permutation network while for hRP, XOR gates are combined to produce the randomized index itself. For the L1 we see that RM outperforms hRP being able to reduce its latency by 2.46×. For the L2, hRP delay is lower than for the L1 since fewer XOR gates are required to produce a wider cache index. However, hierarchical implementations do not necessarily reduce the delay since, despite fewer bits are combined to produce the random index, since this index has fewer bits, more XOR gates are required to produce the output. While in our implementation of the hierarchical approach the two effects compensate each other, thus making latency remain the same, different implementations may provide slightly different results.

Overall, the hierarchical implementation (hRP+MOD/RM) decreases area overheads and reduces the number of critical paths ($\approx 3X$), which in this case correspond to the index bits, w.r.t. the hRP implementation. The latter significantly mitigates the impact that process variations have on the maximum achievable frequency [33]. In particular, for hRP+MOD, the L1 access latency slightly increases by two XOR gates w.r.t. modulo placement. For the L2, hRP+MOD causes a larger impact on critical path due to the higher complexity of its design (XOR gates and barrel shifters). Still, this impact was not enough to decrease the maximum operating frequency.

4.3.2.3 Performance Validation

To validate performance results of the hRP+MOD setup, we run the EEMBC automotive benchmark suite in the FPGA prototype. Our platform does not implement a floating-point unit so we excluded those benchmarks using FP operations. Also, MediaBench requires some I/O RTOS support that is not yet available for this particular configuration, so we did not include them. We made 500 runs of each benchmark and averaged hit ratios, and compared the implemented hRP+MOD and the default MOD against results in the simulator. Results (not shown for space constraints) reveal that the FPGA implementation of hRP+MOD shows almost the same behavior observed in the simulation evaluation. Hit rates are quite high for all EEMBC, specially for the L1 but also for the L2, proving the effectiveness of these placement policies.

4.3.2.4 Average and Worst-Case Performance Results

The first bar in each pair in Figure 4.2 shows that hRP+MOD achieves very similar average performance to that of MOD: 1% worse on average and up to 3%, making hRP+MOD very competitive in terms of average performance. For WCET estimation, we build on current industrial practice for WCET analysis on real boards that takes as WCET estimate a margin (e.g. 20%) over the high watermark (HWM) execution time [153]. The second bar in each pair in Figure 4.2 shows the pWCET estimate obtained for hRP+MOD w.r.t. to the highest execution time observed for MOD (i.e. the HWM). For all benchmarks we observe that the pWCET estimate is above the HWM (as expected) and for all benchmarks but one the pWCET estimate with hRP+MOD is below HWM+20% obtained for MOD. Hence, hRP+MOD helps reducing WCET estimates w.r.t. current practice while increasing the confidence on estimates w.r.t. just increasing the HWM by a fudge factor of 20%. On average, pWCET estimates are just 8% over the HWM (12 percentage points below HWM+20%).

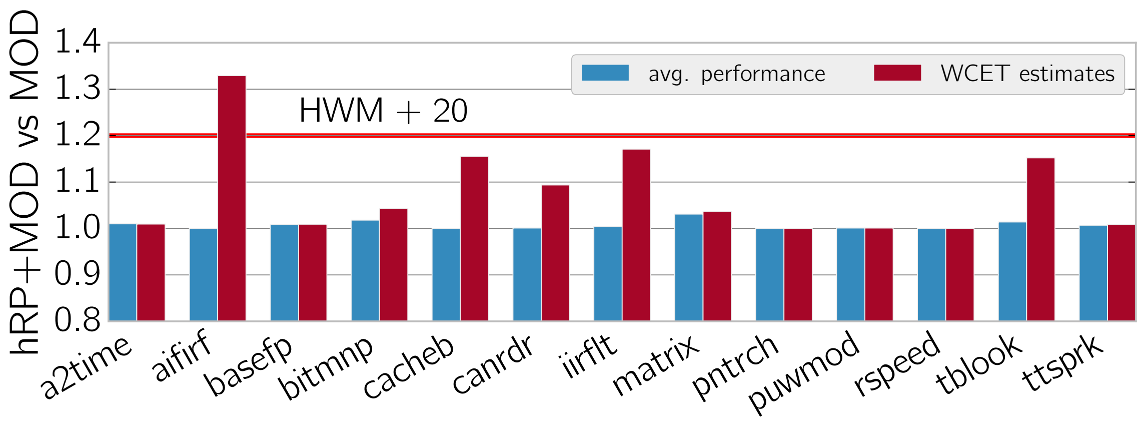


Figure 4.2: Results from the chosen random hardware placement policy compared with modulo placement

4.4 Related Work

In [133] authors propose a pseudo-random hash function to distribute the data across sets and thus, make the cache performance less sensitive to different placements compared to conventional modulo placement. Topham [150] also explores different pseudo-random hashing functions to reduce conflict misses. With skewed associative caches [32], each way uses a distinct hash function for randomized placement across banks, which reduces conflict misses for programs that process large matrices. In [13] the authors study the performance impact different cache placement parameters have on the real-time domain. A commonality of these solutions is that placement uses only the address of the access. As a result, for a given memory layout a single placement is produced across all runs of the program. This poses the same limitations for MBPTA as conventional deterministic architectures based on modulo placement: time composability is lost since performance changes arbitrarily if memory addresses change upon integration.

4.5 Conclusions

While cache memories (in particular multi-level cache hierarchies) offer benefits for RTES, they challenge timing analysis. Some studies show that MBPTA combined with time-randomized caches facilitate factoring in the impact of caches in WCET estimates. However, those studies mostly focus on single-level cache hierarchies. We propose several multi-level configurations and implement them on a simulator. These configurations include both, monolithic and new hierarchical solutions. Finally, we implement the most cost-effective hierarchical configuration in an FPGA, and compare it against a conventional deterministic cache. Our results show that this solution results in negligible average performance degradation and improved (reduced) WCET estimates, while preserving time composability.

Chapter 5

Cache replacement policies

5.1 Introduction

The impact of caches on execution time is highly program-dependent. For instance, some programs barely exploit cache space. In the context of Critical Real-Time Embedded Systems (CRTES), programs may be produced by means of automatic code generation tools (e.g. SCADE [146]), typically leading to programs with few thousands of instructions. For these programs, Random Replacement (RR) (as described in Section 2.2) may produce a first pathological scenario (*ps*1) in which few cache lines fitting in a cache set, randomly evict each other on each miss despite there are some available lines in that set. This occurs because nothing prevents unfortunate replacement choices whose probability decreases, but still must be accounted for with Worst-Case Execution Time (WCET) estimates, which may be relatively high. RR can also cause a second pathological scenario (*ps*2) resulting in the replacement of cache lines that have been just accessed, losing some temporal locality. Both effects have the same root cause: cache lines recently fetched can be randomly evicted shortly after fetched. In order to tackle this challenge, we make the following contributions:

1. We make an in-depth analysis of pathological behavior of RR in terms of probability of each pathological scenario and its potential impact on performance.
2. We propose Random Permutations (RP) and Non-Most Recently Used Random Permutations (NMRURP) that restrict random choices to prevent pathological cases. RP evicts *all* lines in a set, yet in a random order, before it starts evicting occupied lines, whereas NMRURP additionally prevents the last cache line accessed from being replaced. Both techniques reduce the number of evictions that can occur before all lines are placed in a cache set (*ps*1) and increase temporal locality (*ps*2) by reducing the chance of evicting recently accessed lines.
3. We perform a detailed analysis of the suitability of the presented replacement policies, both deterministic policies and their random counter-parts, with respect to the properties needed by Measurement-Based Probabilistic Timing Analysis (MBPTA).

4. Finally, we assess RP and NMRURP complexity and benefits with the Mälardalen benchmarks as well as a railway case study, providing evidence of their feasibility and gains in terms of WCET reduction.

5.2 Analysis of Replacement policies

Several replacement policies have been proposed over the years. We classify them into two main categories: those with a fully deterministic behavior and those with a randomized behavior.

5.2.1 Deterministic Replacement Policies

We analyze Least Recently Used (LRU), Non-Most Recently Used (NMRU) and Binary Tree (BT) replacement policies as a representative of deterministic policies. It is noted that the latter two have been proposed to reduce the hardware requirements of the former [97]. Our analysis shows that all of them have systematic pathological cases, i.e. addresses sequences for which they result in evictions that occur systematically.

5.2.1.1 LRU

LRU keeps the order in which lines in the set have been last accessed, from the most recently used (MRU) to the (LRU). Upon a cache hit, the cache line accessed is promoted to the first position in the list (MRU). Upon a cache miss, the last cache line in the list (LRU) is replaced and used to store the newly fetched cache line, which is then promoted to the first position in the list.

5.2.1.2 NMRU

NMRU can be seen as a simplified version of LRU. Keeping the full order of cache lines in a set is increasingly costly for high associative cache memories: for an N-way cache, LRU requires keeping the full order across the N elements. NMRU instead only prevents the MRU line from being evicted, without imposing any particular order on the other cache lines. This can be achieved, for instance, with a single pointer indicating the next line to be evicted in the set (e.g. in a round-robin fashion) skipping the MRU line if it is selected for eviction. In general, such a solution may not preserve temporal locality as much as LRU, since the 2^{nd} MRU line could be evicted instead of the LRU line on a miss. On the positive end, NMRU scales to arbitrarily highly associative caches with limited cost: a pointer to the next line to be evicted (incremented upon an eviction) and another pointer to the MRU line to protect it from being evicted.

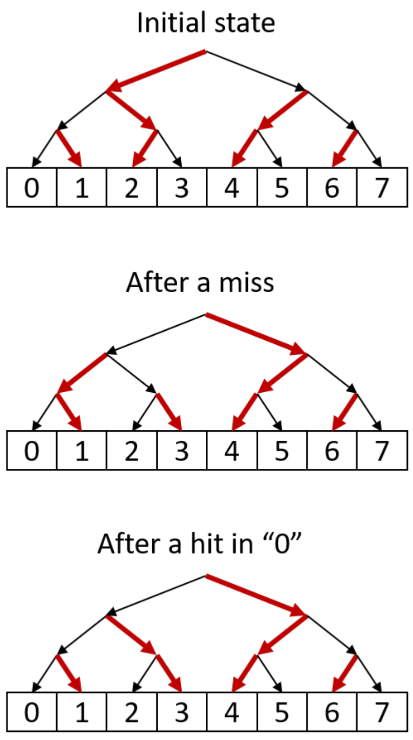


Figure 5.1: Schematic of BT placement operation

5.2.1.3 BT

BT protects the MRU line, as NMRU does, but also keeps some partial order on the remaining lines. Hence, BT lays in-between LRU and NMRU in terms of ability to preserve temporal locality and complexity. In particular, BT partitions cache lines in a set into two halves (left and right) recursively indicating in each partition the side from where to select the line to be evicted. This is illustrated in Figure 5.1 for an 8-way cache set. In the figure, cache line 2 is the candidate to be replaced. Upon an access, all arrows in the path to the cache line accessed are moved away. For instance, as shown in the figure, on a miss all arrows pointing to 2 are changed, thus pointing now to 4. Then, upon a hit on cache line 0, only the arrow pointing to the pair 0-1 is changed to point to the pair 2-3. Note that within the pair 2-3, the corresponding arrow points to 3, thus providing some temporal locality protection for 2, which has been accessed recently. While such protection does not guarantee full order as in the case of LRU, it provides some further temporal locality protection than in the case of NMRU. BT has a logarithmic cost on the associativity, thus limiting the overhead w.r.t. LRU, although it is slightly higher than that of NMRU. In particular, it requires 1 bit for each left/right choice. Hence, N-1 bits are needed per set for an N-way cache.

5.2.1.4 Pathological cases

All deterministic policies have, at least, one pathological case for which accesses systematically result in misses. One such cases for all deterministic policies above is an access sequence with N+1 different addresses accessed in a round-robin fashion.

For instance, for a 4-way cache, a pathological case would be repeating the sequence $ABCDE$ an arbitrary number of times.

- LRU: after the first 4 accesses, A is the LRU line, so E evicts A. Then we access A, which is a miss and evicts B. Then access B results in a miss that evicts C, and so on and so forth.
- For NMRU it can be seen that 5 different addresses are enough to evict cache lines in the 4 sets in a round-robin fashion. In this particular example we never try to evict the MRU line, so the behavior is analogous to a FIFO policy.
- In the case of BT, although less obvious, after 4 misses, the tree points to the first line fetched out of those for replacement systematically, so E evicts A, A evicts B, B evicts C and so on and so forth, analogously to the case of LRU and NMRU.

While our analysis is limited to a subset of replacement policies, it serves the purpose of illustrating that systematic pathological cases exist for deterministic policies due to their intrinsic deterministic nature. Such systematic cases can only be broken, in general, by means of some form of random choice. For instance, this is the case of RR, which randomly selects the cache line to be evicted in the set upon a miss.

5.2.2 RR

The basics about RR have been introduced in Section 2.2. Now we focus on how RR behaves with the pathological cases under study.

RR evicts a specific cache line with a probability of 1/N for an N-way cache. Hence, the probability of a line surviving an eviction is (N-1)/N, and hence, there is a non-null probability of survival for cache lines for any access sequence. In particular, for the systematic pathological case above for deterministic replacement policies, RR provides a survival (hit) probability of $\left(\frac{3}{4}\right)^4 = 0.316$.

On the other hand, if we consider a sequence with N addresses instead of N+1, then we realize that all deterministic policies would lead to all-hit sequences except for cold misses. Instead RR has non-null probability of evicting some cache lines before (randomly) placing all addresses in distinct physical cache lines so that all remaining accesses become hits. This occurs because RR is unaware of temporal locality since it does not preserve any history.

Since Probabilistic Worst-Case Execution Time (pWCET) estimates with MBPTA need to account for the worst case that can occur with non-negligible probability, the pathological cases of deterministic policies need to be accounted. In the case of RR, some low-probability high-execution time eviction scenarios need to be accounted for, but those scenarios will not include the absolute worst case, which is the actual case for deterministic policies under systematic pathological cases. Still, the fact that these eviction sequences can occur under RR may lead to high pWCET estimates, which relates to the fact that RR is a locality-unaware replacement policy. This is illustrated with the following examples, which we refer to as pathological scenario 1 and 2 respectively (ps1 and ps2 for short).

5.2.2.1 *ps1* - The number of objects mapped to a cache line is smaller than or equal to W

Let us consider a fully-associative data cache with 4 ways ($W = 4$) and a program accessing alternatively addresses A and B, which belong to different cache lines, 20 times each. First, A is placed in a random line. Then, B will be placed in a random line, with a probability of 3/4 of not replacing A and 1/4 of replacing it. If B replaces A, then A has a 1/4 probability of replacing B again. And they can keep replacing each other with probability 1/4. Overall, this program experiences $M + 2$ misses (2 cold misses are always experienced), where $M \geq 0$, with a probability:

$$P(M) = \left(\frac{1}{4}\right)^M \times \frac{3}{4} \tag{5.1}$$

For instance, $P(4)$ (the probability of having $4 + 2 = 6$ misses) is ~ 0.003, $P(10) \approx 7 \cdot 10^{-7}$ and $P(20) \approx 7 \cdot 10^{-13}$ (see blue line in the top chart of Figure 5.2). Assuming 10 cycles per miss and 1 cycle per hit, and considering only the impact of cache in execution time, the pWCET at an exceedance threshold of 10^{-12} per run can only be at least 238 cycles to account for 22 misses and 18 hits, since the probability of having at least 22 misses (the accumulated probability of [22,40] misses) is $\sim 9 \cdot 10^{-13}$, see blue line in the bottom chart of Figure 5.2.

5.2.2.2 *ps2* - The number of objects mapped to a cache line exceeds W

Another pathological case arises when the number of objects mapped to the same set exceeds W. In this case, RR can lead to the loss of temporal locality, since random eviction patterns can make recently touched objects (i.e. cache lines) be evicted from cache on a miss, whereas old-standing (unlikely to be reused) objects remain in cache. For instance, in the repeating sequence $ABACADAE...$, where A is continuously interleaved with accesses to addresses that lead to a cold miss, RR may evict A sometimes. Conversely, deterministic policies presented in previous section would always protect A from eviction since it is the MRU line upon the access to any other address. As the number of objects mapped per set increases, cold and capacity (unavoidable[1]) misses become the main contributors to miss rates, naturally reducing the benefit of any replacement policy.

5.3 Locality-aware RR

In this section we introduce our proposed (temporal) locality-aware RR policies. In particular, we propose RP and NMRU NMRURP, which aim at avoiding systematic pathological cases such as those of deterministic policies as well as preserving higher levels of temporal locality than RR, thus improving RR.

[1]They could only be avoided prefetching cache lines, but still prefetch requests would need to fetch data from upper memory levels.

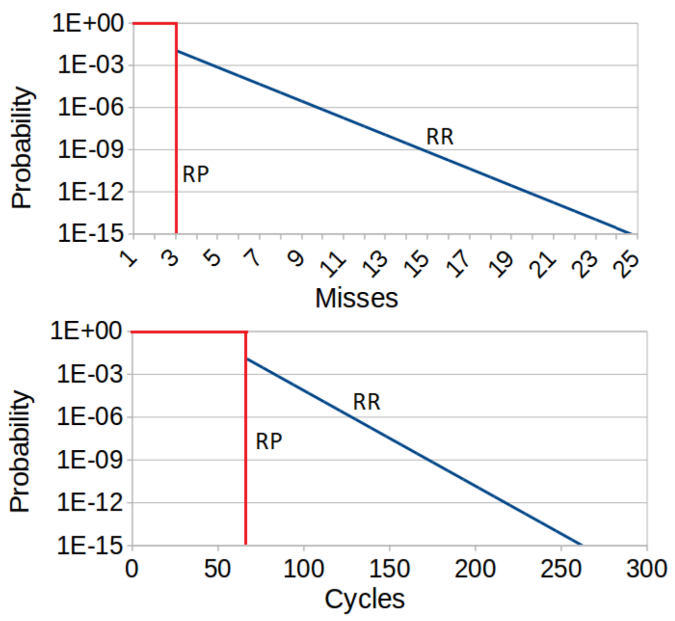


Figure 5.2: Comparison of pWCET and probabilistic miss count curves between RR and RP

5.3.1 RP

RP limits pathological RR scenarios by increasing temporal reuse and enforcing random evictions to occur across all cache ways.

- When accessed data fits in a cache set, they will eventually be placed in different cache lines, thus avoiding potentially long mutual evictions by construction. This would result in a single replacement for the previous example, thus leading to a maximum execution time of 67 cycles (3 misses and 37 hits), see red lines in Figure 5.2.
- When the number of accessed lines exceeds the size of a set, RP effects are also positive increasing reuse, though the impact of replacement naturally reduces.

To reach its goals, RP leverages the concept of RP [82], which we first introduce and then explain how can be used and implemented in RP. We finally show how the resulting RP limits pathological eviction patterns.

5.3.1.1 Logic Behind RP

RP avoid potentially infinite starvation in the arbitration for shared resources, where one requester (unluckily) loses all arbitrations with decreasing probabilities. This occurs with standard random (lottery) arbitration [98], with which on every arbitration round the grant is randomly given to one of the requesters without taking into account how long requests have been waiting. Hence, while each of the N_r requesters is granted access $1/N_r$ of the times in the long run, one requester could suffer long starvation periods.

Instead, RP generates in every *arbitration (or permutation) window* a random permutation of all potential requesters. While the particular requester that is granted access in each arbitration round is random, each of the N_r requesters is granted access exactly once every arbitration window. For instance, for a resource shared across $N_r = 3$ requesters ($r1$, $r2$ and $r3$) each requester has 1/3 chances to be granted access first. If $r2$ is granted access, then $r1$ and $r3$ have 1/2 chances to be granted access second, whereas $r2$ cannot be granted access second. If $r3$ is granted access second, then $r1$ is automatically granted access third.

RP is implemented by creating a list where each requester appears once and sorting it randomly. Note that a requester could request access to the shared resource at any point in time w.r.t. the current arbitration window. Thus, the worst case occurs when, for instance, $r2$ arrives in the second slot of the current arbitration window, $r2$ was the first one in the window (so it just missed its opportunity), and has to wait for its slot in the next window, which, potentially, can be the last one. Recalling the example before, we could have the following arbitration windows: $< r2, r3, r1 >$, $< r3, r1, r2 >$, and $r2$ could arrive right after its slot in the first window has elapsed. Overall, in general the maximum number of slots a requester may have to wait is $2 \cdot (N_r - 1)$. This limits how long a request waits to be granted access.

RP can be applied with the same logic to the replacement policy. Instead of randomly choosing the way within the cache set that will be evicted next, RP generates a *random permutation window* per set in which the number of elements matches the number of cache ways W. Whether a line is evicted next is a random event (each line can be evicted with $1/W$ probability), but the eviction choices, though random, are not independent among them. In other words, each different permutation has the same probability to be created, each way number is in each of the W positions of the N_{Perm} different permutations N_{Perm}/W times and permutations are chosen (generated) randomly. However, given that each permutation contains each way number exactly once, it is impossible that a way is not selected for more than $2 \cdot (W - 1)$ evictions, and a particular way can be evicted at most twice consecutively (if it is the last in one permutation and the first in the following one).

5.3.1.2 Implementing RP

For a W-way cache the total number of potential permutations of cache ways is $N_{perm} = W!$. At hardware level, implementing such an ideal solution could require $W! \cdot \lceil log_2(W) \rceil \cdot W$ bits for the permutations table, $\lceil log_2(W!) \rceil$ bits per set for the pointer that selects the permutation, and $\lceil log_2(W) \rceil$ bits for selecting the current permutation entry, plus the control logic for such an implementation. For a 4-way cache this would mean 192 bits for the table and 7 (5+2) bits per set. In an 8-way cache or higher this number significantly increases. In this section we implement a low-complexity solution that limits the number of potential permutations while preserving the properties of the ideal solution, and matching its average and WCET performance. We use area and logic as main metrics to assess the hardware feasibility of our approach.

Registers area: We implement RP by adding a bit vector per cache set, similar

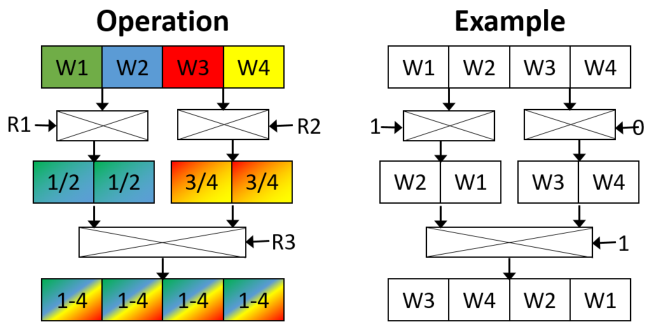


Figure 5.3: Random permutation operation example for a 4-way cache

to that needed for LRU replacement. In the vector each way number is represented exactly once. Given a cache with W ways, this vector requires W fields with $\lceil log_2 W \rceil$ bits each, plus a pointer of $\lceil log_2 W \rceil$ bits to point to the current position in the vector. Thus, a 2-way cache requires 2+1 (vector+pointer) bits, a 4-way cache 8+2 bits and a 8-way cache 24+3 bits. For comparison purposes, LRU requires the same number of vector bits per set to keep the eviction order. Hence RP has similar area requirements in terms of bits to keep the replacement state as LRU, and only adds the bits of the pointer indicating the current position in the permutation.

Additional logic: The vector part of RP for a 4-way cache is depicted in Figure 5.3 (left). We denote the id assigned to cache ways as $w1$, $w2$, $w3$ and $w4$ respectively. Whenever the pointer wraps up, a new random permutation is generated. This is done, as shown in the picture, swapping different parts of the vector based on some random bits: $R1$, $R2$ and $R3$. $R1$ determines whether the first two elements are (randomly) swapped or not. $R2$ does the same for the last two elements. Finally, $R3$ determines whether the first pair of elements is swapped or not with the second pair. This simple implementation allows to generate a new permutation quickly and efficiently. LRU, instead, needs being able to extract any element from the list, place it at the beginning and shift all leftmost elements one position ($\lceil log_2 W \rceil$ bits) right. Thus, multiplexers (as for RP) and expensive parametric shifters are needed for LRU which compromises its scalability.

Random bits can be easily generated with a single low-cost linear feedback shift register [9, 122], which meets the requirements of MBPTA. Although each cache set has its own random permutation, the number of new permutations needed in one cycle in the cache is, at most, as many as cache ports exist since, in the worst case, all simultaneous accesses could produce a miss that requires a new random permutation in their respective cache sets. This would require 3 random bits per cache port simultaneously. The PRNG used has been proven capable to produce at least 32 bits per cycle if needed, thus making a PRNG able to feed multiple caches. On average, however, each cache port will require one new permutation every W cache misses, which occur seldom. Thus, usual random bit requirements per cycle are very low and a single PRNG could fit all caches in one or several cores.

One feature of our implementation is that it generates a subset of all potential permutations (8 of the possible 24 in a 4-way cache). For instance, $w1$ and $w2$ can never be in separate pairs. This means that permutation $< w1, w3, w2, w4 >$ could never be produced. Yet our implementation still preserves the properties needed by MBPTA on RR: a given way occupies each position in the permutation with identical probabilities and where they are allocated is a purely random choice. The right side of Figure 5.3 shows an example of the generation of a new random permutation for the replacement of one cache set. Initially, we have the permutation $< w1, w2, w3, w4 >$. Given that random bit $R1 = 1$, $w1$ and $w2$ are swapped. Since $R2 = 0$, $w3$ and $w4$ are not swapped. Finally, $R3 = 1$, so $< w2, w1 >$ and $< w3, w4 >$ are swapped, leading to the new permutation $< w3, w4, w2, w1 >$.

In Figure 5.4 we compare the implementation costs of the fully randomized design and our cost efficient one of RP. For caches with 2 or 4 ways, the implementation cost is roughly the same. However, from 8 ways to 32 the cost of the ideal solution requires a million to 10^{37} bits respectively, while the efficient solution only needs 2000 to 10000 bits.

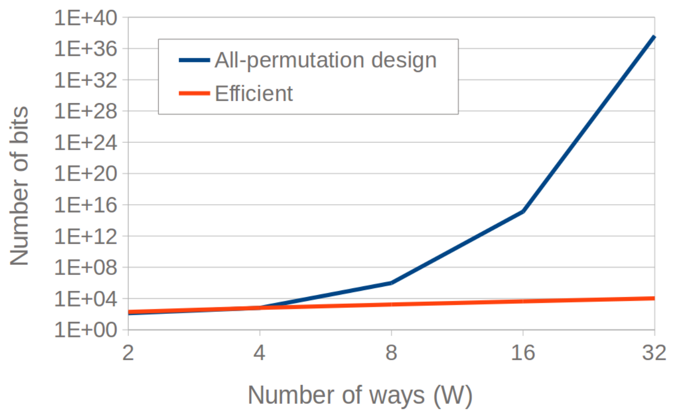


Figure 5.4: Hardware implementation cost for a 64-set cache of random permutation for different design choices

5.3.1.3 Controlling Pathological Scenarios

RP controls the pathological scenarios drawn for RR, i.e. ps1 and ps2, as presented next.

ps1: when W or fewer lines are (randomly) placed in the same set, they can replace each other a limited number of times. Let us recall the example in Section 5.2 where addresses A and B are accessed repeatedly. With RP two scenarios can occur (illustrated in Figure 5.5):

1. A and B are granted access with the same permutation window to take eviction decisions. In this case, A and B will be placed in different random ways.
2. A evicts a line using the last element of one permutation window and B uses the first element of a new permutation window. In this case, again, two scenarios can occur. Firstly, A and B use different ways. And second, A and B are

randomly mapped to the same way. In the latter case, B evicts A, but next time that A is fetched will necessarily use the same permutation window as B, so it will be placed in a different way. Thus, at most one mutual eviction will occur.

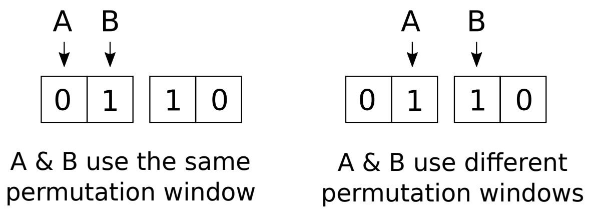

A
B
0 1 1 0
A & B use the same permutation window
A
B
0 1 1 0
A & B use different permutation windows

Figure 5.5: Example of the two different scenarios that can occur in pathological scenario 1 with RP

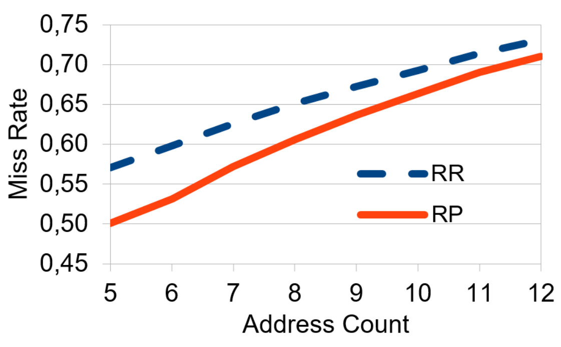
Overall, when 2 different addresses compete for the space in a given cache set, at most 1 mutual eviction will occur. In the general case, if K different addresses are accessed, where $K \leq W$, the worst case occurs when $K-1$ addresses use the last elements of a permutation and the last address uses another permutation so that it evicts one of the other $K-1$ addresses which, in turn, evicts another and so on and so forth. However, eventually the K addresses produce K consecutive evictions using elements of the same permutation, thus being placed in different cache ways and avoiding pathological evictions. Thus, the maximum number of pathological evictions can be expressed as:

$$N_{maxevict} \leq K-1, \forall K \leq W \tag{5.2}$$

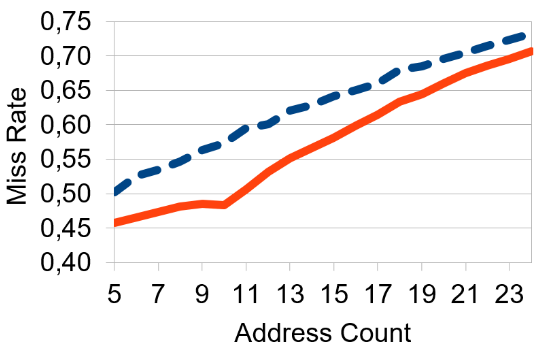
ps2: when the number of addresses mapped to a set exceeds the number of ways, i.e. $K > W$ compete for the same cache set, on every miss, RR can randomly evict recently touched lines hence decreasing temporal reuse. To show this we run an experiment in which we access a given number of addresses K in a sequence of size $2 \times K$ in which the addresses are randomly selected. We assume that all addresses are mapped to the same set and analyze the average hit rate of RR and RP when processing the same random sequence 1,000 times. Figure 5.6 shows the hit rate, for a 4-way cache (e.g. DL1-like) and a 8-way cache (e.g. L2-like), of RR and RP as K varies from $W+1$ to $W*3$. We see that RP consistently lowers miss rates for both setups, with the benefit decreasing as the number of addresses increases w.r.t. the way size W since the overall miss rate naturally equalizes for high address counts, i.e. $K >> W$, when cache set capacity is largely exceeded.

5.3.2 NMRURP

NMRURP, in the same line as RP, limits pathological RR scenarios by increasing temporal reuse and enforcing random evictions to occur across all cache ways, but additionally, it guarantees that the MRU line cannot be evicted. In particular, it avoids potentially long mutual evictions by construction, thus behaving as RP in the



(a) DL1 ($W = 4$).



(b) L2 ($W = 8$).

Figure 5.6: Miss rate as a function of the number of accessed addresses for cache sizes similar to L1 and L2 respectively between RR and RP

example in Figure 5.2. Whenever the number of cache lines exceeds the space of the corresponding cache set, NMRURP has still some positive effects on reuse, but obviously the impact of replacement policies diminish as the set capacity is increasingly exceeded.

In order to introduce NMRURP, we build upon our other proposed randomized replacement policy, RP. First, we show how in specific cases RP may not favor locality sufficiently, and then how NMRURP improves over RP.

5.3.2.1 Locality Awareness of RP

RP improves locality over RR by avoiding the replacement of a given cache line within a permutation (window). Hence, cache lines recently fetched cannot be evicted before crossing the boundary to the next window. However, there are two scenarios where RP may fail to preserve locality:

1. RP places no constraint across arbitration window boundaries. Hence, potentially, the same physical cache line could be evicted twice consecutively, which would allow the MRU line to be evicted. This occurs whenever a cache line is the last in a window and the first in the following window.
2. Also, RP does not keep any history on whether cache lines have been hit recently. Hence, if a given cache line can be the candidate for replacement according to RP, be hit, and then be evicted immediately due to a miss, thus causing an eviction of the MRU line.

Next, we illustrate those two scenarios with specific examples that serve the purpose to motivate the introduction of NMRURP.

RP example 1: window boundaries. Let us assume a 4-way cache whose current and next arbitration windows are $< w1, w2, w3, w4 >$ and $< w4, w2, w3, w1 >$ respectively. Let us further assume that the next eviction is dictated by the last slot of the current window ($w4$ in the first window), and $w4$ in the cache set contains cache line A. The sequence $B_1A_1B_2$, where the subscript only indicates access ordering to a given address, would cause 3 misses. First, B_1 would miss and would evict the line in $w4$, so address A. The pointer in the arbitration window would move to the first position in the next permutation, which is $w4$ again. Then A_1 would also miss, thus evicting B. Finally, B_2 would also miss and would replace the address in $w2$.

As shown, RP allows evicting the MRU cache line even if it has just been fetched when crossing arbitration window boundaries if the last slot of one window and the first slot of the following window randomly point to the same cache way, which occurs with probability $1/W$, where W is the number of cache ways.

RP example 2: MRU hit. Let us assume the same example, so the current arbitration window is $< w1, w2, w3, w4 >$, the candidate for eviction is $w4$, and A is stored in $w4$. In this case, the sequence $A_1B_1A_2$ would produce 2 misses since A_1 is a hit, thus becoming $w4$ (so A) the MRU line, B_1 is a miss and evicts A, and then A_2 also misses.

As shown, on a hit there are $1/W$ chances that the candidate for eviction is the cache line recently hit, so a subsequent cache miss may evict the MRU cache line with a non-negligible probability.

5.3.2.2 NMRURP Replacement Policy

NMRURP is a hybrid policy between NMRU and RP. In particular, it works as RP but, whenever the cache line to be evicted is the MRU, the following candidate in the arbitration window (or the first one in the next window if window boundaries are exceeded) is selected for eviction. This prevents, by construction, the eviction of the MRU cache line.

Let us recall **RP example 1** above. In this case, B_1 would evict A from $w4$, but A_2 would not be allowed to evict B since B is stored in the MRU way ($w4$). Hence, $w2$ would be replaced instead and access B_2 would hit. In the case of **RP example 2**, the behavior is similar. A_1 hits in $w4$, B_1 is not allowed to evict $w4$ and evicts $w2$, and A_2 is therefore a hit.

Overall, NMRURP increases locality w.r.t. RP in both cases, whenever the MRU was either hit or fetched due to a miss.

5.3.2.3 Implementing NMRURP

Implementation costs of NMRURP are slightly higher than those for RP. In particular, it requires a register of $\lceil log_2 W \rceil$ bits to store the identifier of the MRU cache line, two comparators to compare the current and next candidates for eviction with the MRU, and a priority decoder to select the appropriate candidate for eviction. Note that the output of the comparisons drive both, the priority decoder and the pointer shift in the arbitration windows to select the following eviction candidate. A schematic of the additional logic w.r.t. RP is depicted in Figure 5.7 for illustration purposes. As shown, only two slots need to be compared with the MRU since at most two consecutive slots may point to the same cache way given that each way has only one occurrence per window, and thus they can only repeat once across window boundaries. Thus, up to two slots may need to be bypassed, as it would be the case in **RP example 2** above.

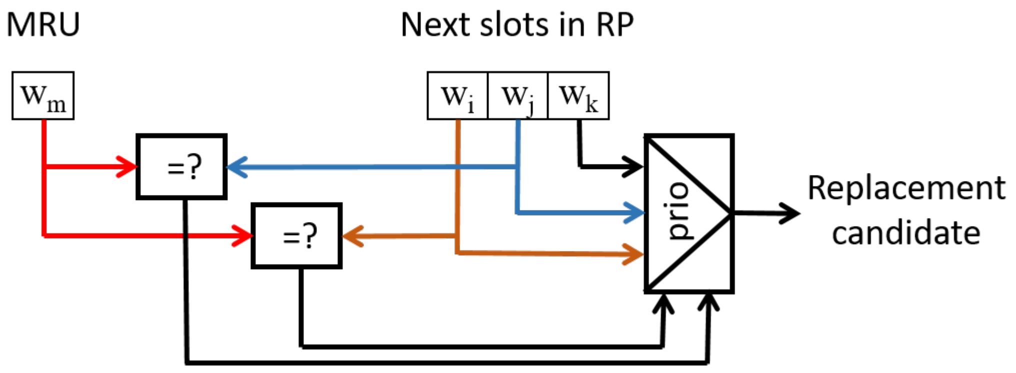


Figure 5.7: Additional logic for NMRURP with respect to RP

Overall, the additional hardware cost is small and increasingly associative caches only require a slightly larger MRU pointer, thus justifying the scalability of this replacement policy.

5.4 MBPTA compliance

MBPTA requires that execution time distributions occurring during operation are matched or upper-bounded by those enforced at analysis. This is achieved by means of time randomization or time upper-bounding [96], as previously explained in Section 2.1.4. In the particular case of replacement policies, this needs to be enforced too. In this section we review the MBPTA compliance of the different replacement policies discussed in this Chapter, namely LRU, NMRU, BT, RR, RP and NMRURP.

5.4.1 Cache Controllability

In general, it is unaffordable predicting the cache state before running a program during operation, unless an explicit cache flush command is executed right before accessing the cache. Flush commands, however, can only be executed at specific time points due to the inability of the Real-Time Operating System (RTOS) to intervene in-between each software unit, the intrusiveness of those interventions, and the impact on time and energy consumption of flushing cache contents often. Hence, cache flushing often occurs only across time partitions for the sake of memory consistency, but not across software units runs within a given time partition [153].

In this context, the most convenient solution to ensure worst-case cache effects are properly captured consists of enforcing an initial cache state at analysis time that leads to equal or higher execution times than any potential initial cache state that may occur during operation. In general, one would expect that the empty cache state provides this behavior, since no data is reused from previous runs and, consequently, more accesses should miss in cache. However, as shown in this section, this is not always the case for all replacement policies.

5.4.2 LRU

The LRU policy keeps the order in which the cache lines in a set have been last accessed. Therefore, two setups starting from different initial cache states will reach the same state for a given cache set after W accesses to different cache lines in that cache set, where W stands for the number of cache ways.

This occurs because, whenever a cache line is accessed, whether it is a hit or a miss, it is promoted to the MRU position, and the remaining cache lines are shifted as needed to the LRU position to make room for this line be the MRU. If the access is a miss, all lines are shifted one position closer to the LRU position, and the LRU line is evicted. Conversely if the access is a hit, only those lines closer to the MRU position than the one hit are shifted towards the LRU position.

The only difference across two different initial cache (set) states with LRU relates to whether the first access to each of the first W different addresses accessed is a hit or a miss, which depends on the initial state.

Therefore, we can upper-bound the behavior during operation by doing one of the following things at analysis:

- Flush the cache before each run at analysis. This ensures that the first w accesses to different addresses in each set miss, and after those accesses the cache state is the same as that during operation regardless of the initial state during operation.
- Add the latency of w misses per set to the WCET estimate, which upper-bounds the gain obtained between the best and the worst initial cache states. This method could result in a more pessimistic outcome since we could be accounting for some gains that do not occur in practice because the program could also miss in the first w accesses to different addresses in each set.

If either of these two approaches is followed, the execution times of the program at analysis with the LRU replacement policy upper-bound those during operation and hence, LRU is MBPTA compliant. Note, however, that despite LRU its compliance with MBPTA requirements, it may still exhibit systematic pathological cases that, nevertheless, would already be captured during the test campaign at analysis.

5.4.3 NMRU

NMRU policy selects the replacement based on a round-robin mechanism but protecting the MRU cache line in a set. Given two initial cache states, the MRU value in a specific set will be the same after just one access. The rest of the addresses, however, will depend on the initial state and the access sequence.

The example in Figure 5.8 shows the same set for two different initial cache states c_0 and c_1. c_0 corresponds to the empty state, whereas c_1 has some contents. The pointer in each set indicates the next element to be replaced (following a round-robin policy). The bold cache line indicates the MRU line, so the one protected from eviction.

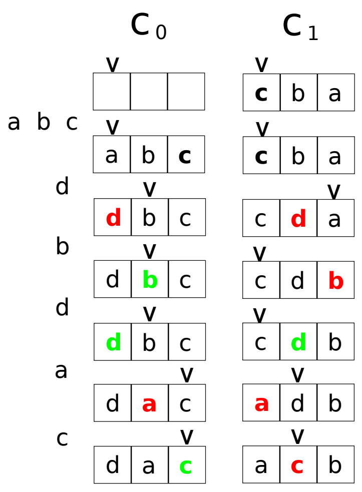


Figure 5.8: Pathological scenario of NMRU: sequence that has more hits in the empty cache than in the one initialized

In this example, first, we make three accesses to fill the empty cache: a, b and c. Afterwards, we access a new cache line d that misses in both caches. Since c_0 and c_1 have different pointers and orders, they will evict different lines. c_0 will evict line a, whereas c_1 will evict line b, since the eviction pointer points to c that is protected (MRU line). The next access is to line c, which hits in c_0 and misses in c_1. After two more accesses that hit and then miss in both caches, the same case arises: an access to line c that hits in c_0 and misses in c_1.

As shown in this example, there is no guarantee that an empty initial cache state upper-bounds the execution time of a non-empty cache, and the execution time difference can be arbitrarily large since both initial states may not converge to the same state. Hence, we can claim that the empty cache state is not an acceptable initial state for analysis runs since it does not upper-bound all initial cache states and access sequences. Moreover, our example already illustrates that specific access sequences may make any given initial state perform worse than another given state without converging to the same state, thus indicating that MBPTA compliance is not achieved for NMRU.

Only if we could enforce the same initial cache state at analysis and during operation, NMRU could be made MBPTA compliant. However, as explained before, the initial cache state during operation may not be controlled (e.g. flushed) in many cases.

5.4.4 BT

The BT replacement policy has an auxiliary tree structure that defines the state of the replacement algorithm for each cache set. The fact that cache lines are already stored in a particular location and such location, together with the access sequence, determines the replacement order, can lead to a pathological scenario where some accesses always miss. This can easily be seen with an example.

The example in Figure 5.9 shows a set of two different initial cache (set) states

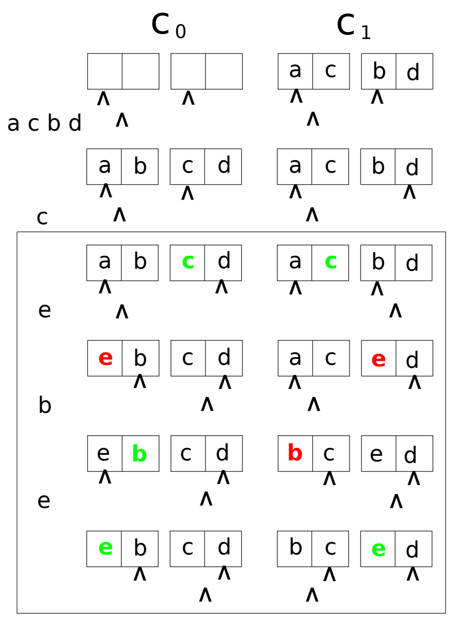


Figure 5.9: Pathological scenario of BT: sequence that has more hits in the empty cache than in the one initialized

c_0 and c_1, one empty and one initialized respectively. The first 4 accesses, namely a, c, b and d are performed in both caches. In this example, the arrows point to the cache line to be replaced. Since the cache has 4 ways, we need 2 levels of arrows. The first level indicates the pair to be replaced first, and the second level what cache line inside the pair must be replaced. After these 4 accesses c_0 is filled (4 misses) and c_1 maintains its state (4 hits).

Another access to c occurs and hits in both caches. Then, we make a sequence of 3 accesses to cache lines e, b and e (all mapped to the same set). The first access misses on both caches, and the last hits on both. However, the second access (b) hits on the empty initial cache state, whereas it misses on the already initialized cache. The final state is equivalent to the third state shown in the example (after accessing c), but with the following conversions: $a' = e$, $b' = c$, $c' = b$, $d' = a$, $e' = c$, where the *prime* mark indicates the new state. This means that a sequence a', c', a' would again result in the same behavior: miss for both, then a hit for c_0 and a miss for c_1, and finally a hit for both. This pattern (shown in a box for illustration purposes), with the appropriate addresses, could repeat an arbitrarily long number of times, thus making the empty initial cache state lead to arbitrarily lower execution times than the non-empty state.

As for NMRU, given any initial cache state, we can devise an access sequence that makes a different initial cache state lead to systematically lower execution times. Hence, an initial state that upper-bounds all others does not exist. This implies that, measurements collected on an empty initial cache state do not upper-bound operation-time behavior, and differences across initial states cannot be upper-bounded in general. Because of this, we regard this cache policy as non MBPTA compliant.

5.4.5 RR

RR policy chooses where to allocate a new cache line randomly. Hence, it does not keep any state on replacement order. With RR we cannot determine how many accesses to different addresses will make two different initial cache states reach the same state. However, since eviction choices are random and have uniform probabilities across lines, we can claim that whether accesses hit or miss does not alter cache replacement state (which is none for RR). Thus, any non-empty state leads, probabilistically, to equal or lower miss rates that an empty initial state.

This behavior can already be inferred from the work in [95]. In particular, authors prove that with RR, given an initial cache state, causing random evictions can only lead to a probabilistically worse execution time. In our case, by causing an infinite number of evictions, we would reach the empty cache state that, therefore, would lead to a probabilistically worse execution time than any other initial state.

Hence, on a non-empty initial cache state during operation, we may experience additional hits and thus, lower execution times than those experienced at analysis with an empty cache state. We can conclude, therefore, that RR is MBPTA compliant with an empty initial cache state at analysis.

5.4.6 RP

RP generates a permutation that will replace all cache lines in a cache set in w replacements. However, a program can start its execution in the middle of a permutation, so there can be a scenario where we need to perform $(2 \cdot w) - 1$ replacements before all cache lines in the set have been replaced. This occurs when a given cache line is in the first slot of a window and in the last of the next window, and initially we start replacing the cache in the second slot. For instance, given the permutations $< w1, w2, w3, w4 >$ and $< w4, w2, w3, w1 >$, if the eviction pointer is in the second slot of the first permutation, we need 7 evictions to evict the line in $w1$, whereas if we are at the beginning of a permutation, we only need w replacements to evict all lines.

Let us build our argument on the MBPTA compliance of RP in two steps. First, we show that the difference between two empty initial cache states with different window alignment is up to $w - 1$ misses. Then, we show that for a non-empty cache state exists an empty cache state that upper-bounds the non-empty state. Such empty cache state is, by construction, up to $w - 1$ misses better than the worst empty cache state. Hence, at analysis we can enforce an empty initial cache state, where window alignment per set can be *any*, and increase WCET estimates $w - 1$ misses per set.

Difference across empty initial states. With RP there is a dependence between the position of the eviction slot of the current window and miss probabilities. In particular, given two initial empty cache states with different permutation window alignment, it may take up to $w - 1$ evictions until their eviction probabilities match (e.g. both align at the beginning of the permutation window), and from that point onwards, eviction probabilities are identical. Whether those up to $w - 1$ replacements lead an additional hit or miss each, depends on the access sequence. Hence, if we increase the WCET estimate by the impact of $w-1$ misses per set, we can claim that RP is MBPTA compliant.

This can be better illustrated comparing RP with NMRU. Since MRU protection is a subcase of LRU, which is MBPTA compliant, let us consider only the round-robin part of NMRU for the sake of this discussion. NMRU fails to be MBPTA compliant because the eviction order of addresses is fixed (round-robin). Hence, the particular location of the pointer in the sets *determines* evictions. In the case of RP, evictions occur randomly and uniformly distributed across cache ways within a permutation window. Once two initial cache states reach the same window alignment, eviction probabilities are random and follow the same distribution across both states, thus having similar properties to those of RR. Hence, reaching such identical alignment (e.g. between the current window alignment and the worst potential alignment for the program under analysis) requires up to $w-1$ evictions.

Difference between empty and non-empty states. Note that since cache hits do not alter RP state, a non-empty initial cache state c_1 can only lead to lower execution times that, at least, an empty initial cache state c_0. In particular, given an access a hitting in a preexisting line in c_1 and missing in c_0, the likelihood of a being evicted in c_0 is lower since the pointer moves to the following slot in the window. Eventually, this may make that a access to a is a hit in c_0 and a miss in c_1, thus producing the opposite effect. However, such extra miss in the non-empty cache state can only occur after an extra miss in the empty cache state, which guarantees that the empty cache state is probabilistically worse than the non-empty one.

Need for padding WCET estimates. So far we have shown that, theoretically, we may need to account for up to $w-1$ extra misses per set. However, this holds under the assumption that the initial alignment is deterministic. However, we can break such dependence by randomizing the initial alignment with the permutation window both at analysis and during operation. We can enforce the flush process to choose randomly the window alignment in each set. At analysis, such flush is performed before each run. During operation, it is only needed at boot time. After that, the random alignment is modified by random choices for replacement, thus leading to a random alignment before running the program under analysis, even if the number of evictions before its execution is deterministic (e.g. programs performing only cold misses). Therefore, RP provides MBPTA compliance by simply starting from an empty cache state with random window alignments in each set.

5.4.7 NMRURP

The case of NMRURP is analogous to that of RP, with the difference that the MRU line is protected. Hence, the difference in terms of permutation window alignments is up to $w-2$ lines. However, the MRU pointer may also differ across different initial states, so the maximum difference across empty initial cache states is again $w-1$ misses per set, as for RP. Also, the same reasoning that applies for empty and non-empty cache states for RP, applies for NMRURP, as well as the concept of enforcing a random window alignment on a cache flush.

Overall, by using an empty initial cache state and enforcing random window alignment in each set on a cache flush (flush must be used at boot time during operation), NMRURP can also be regarded as MBPTA compliant.

5.5 Evaluation

In this section, we evaluate RP and NMRURP in terms of average execution time and pWCET estimates, and compare them against LRU, NMRU, BT (average performance) and RR, LRU (average performance and pWCET).

5.5.1 Framework

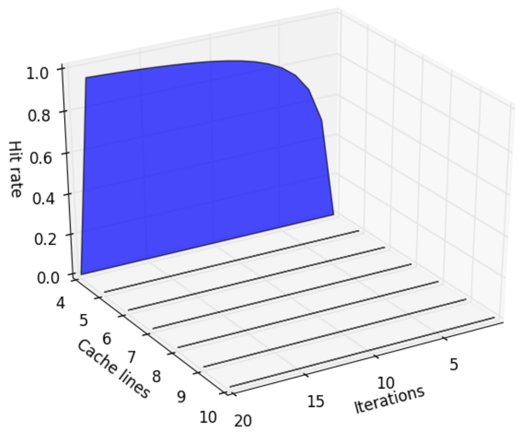
5.5.1.1 Processor model

We evaluate our proposal using the system and simulator described in Sections 3.2 and 3.2.4. L1 caches implement Random Modulo (RM) placement [72], whereas L2 cache implements hash-based random placement [92], given that this has been shown a very convenient setup [72] for MBPTA. Note that, since we build upon a randomized placement policy, execution times change across runs, even if the replacement policy is deterministic as, for instance, in the case of LRU. By choosing a randomized placement policy, we enable MBPTA compliance for all those replacement policies also MBPTA compliant, and have a fair comparison across replacement policies since all of them build upon the same placement policy. In fact, we enforce the same set of random placements (e.g. the same set of 1,000 random placements for each of the 1,000 runs) across replacement policies so that the only source of variation across setups is the replacement policy. For average performance evaluation we consider that all caches use the same replacement policy, which can be either LRU, NMRU, BT, RR, RP or NMRURP. The L2 has no replacement policy since each core has a single L2 way, thus requiring no replacement policy. For pWCET estimation purposes, we compare our proposals, namely RP and NMRURP, with the existing MBPTA compliant replacement policies, namely RR and LRU. Note that, despite considering a multicore, evaluation is performed for programs in isolation since we focus on cache replacement effects.

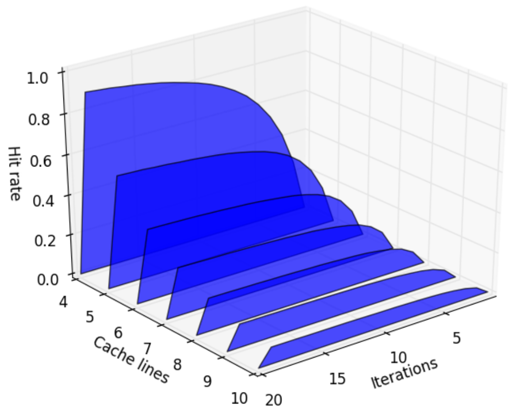
5.5.1.2 Applications

We make a solid evaluation of our proposal with three different application setups.

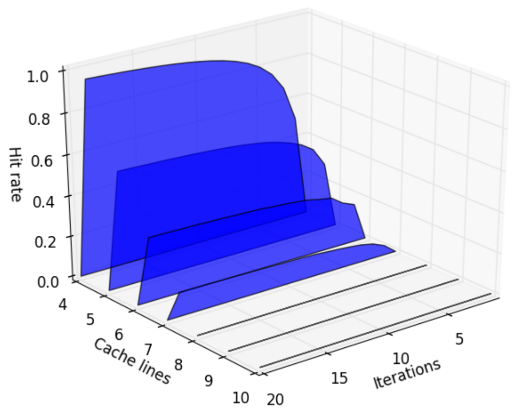
- For illustration purposes, we consider a synthetic benchmark traversing a configurable number of times a vector with varying sizes ranging from 4KB to 40KB in 4KB steps, on a setup with modulo placement, so that we access in a round-robin fashion between 1 and 10 different lines in the same set. This allows us illustrating the different timing behavior for each replacement policy.
- We use a representative subset of the well-known Mälardalen Benchmark Suite [64].



(a) LRU/NMRU/BT.



(b) RR.



(c) RP/NMRURP.

Figure 5.10: Relacement policies result comparison: L1 data hit rate for the synthetic benchmark when varying the number of cache lines and iterations

- We also use a real railway application implementing a critical real-time function from the European Train Control System (ETCS) reference architecture.

5.5.2 Average Performance

For these experiments, we use the same random seeds (and so, the same placements) for all configurations so that differences are produced due to the replacement policy. Miss rates as well as average execution time are obtained as the mean across all measurements for each input set and replacement policy.

5.5.2.1 Synthetic Benchmark

Figure 5.10 analyzes the hit rate (y-axis) of the synthetic benchmark varying the number of cache lines accessed in round-robin (x-axis) and the number of iterations of the loop (z-axis).

- Figure 5.10(a) shows the matching results results for LRU, NMRU and BT replacement. We observe that a very high hit rate is obtained for up to 4 addresses accessed, which matches DL1 associativity. Above that point, all addresses are evicted systematically before being reused.
- In the case of RR (Figure 5.10(b)), we observe that it obtains decreasing hit rates as the number of addresses increases, but they are never zero. However, we also observe that hit rates slowly increase with the number of iterations for 4 addresses due to the cases where lines evict each other despite fitting in cache. A similar trend occurs for 2 and 3 addresses, but it is omitted in the plot since visually it is not so obvious.
- Finally, Figure 5.10(c) shows the matching results for RP and NMRURP. We observe that the hit rate grows rapidly with the number of iterations for 4 addresses. It also grows faster than RR for 2 and 3 addresses. However, for larger address counts the hit rate decreases faster than for RR being zero for 8 addresses. However, in that case the real problem is not the replacement policy, but the fact that cache capacity has been largely exceeded.

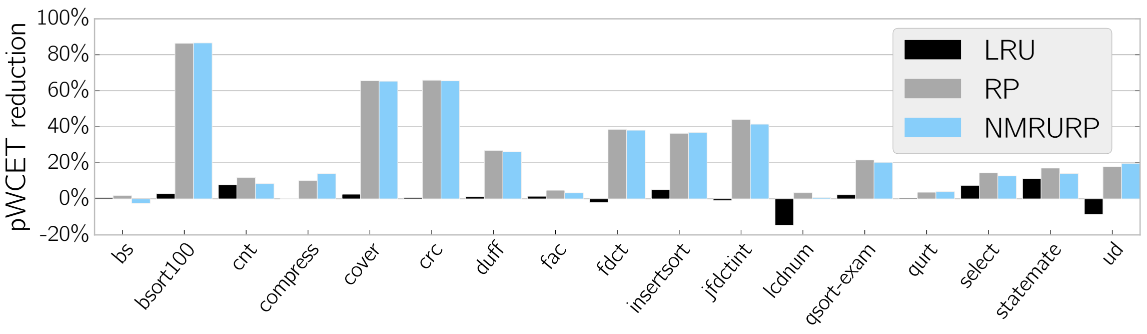


Figure 5.11: pWCET reduction ($p = 10^{-12}$) for the Mälardalen benchmarks w.r.t. RR.

Table 5.1: Replacement policies: average results for the 10,000 executions and the worst 100 (1%) for the Mälardalen benchmarks

	All					
	LRU	**NMRU**	**BT**	**RR**	**RP**	**NMRURP**
Cycles	45522	45036	44776	46076	44998	44989
IL1 m.r.	0.017	0.017	0.017	0.018	0.017	0.017
DL1 m.r.	0.133	0.133	0.128	0.135	0.132	0.131
L2 m.r.	0.465	0.460	0.478	0.459	0.464	0.465
	Worst 1%					
Cycles	46472	45428	45218	46991	45357	45358
IL1 m.r.	0.018	0.018	0.018	0.019	0.018	0.018
DL1 m.r.	0.134	0.137	0.132	0.138	0.136	0.136
L2 m.r.	0.424	0.400	0.405	0.416	0.401	0.401

5.5.2.2 Mälardalen

Table 5.1 shows the average number of cycles and miss rates for IL1, DL1 and L2 for the 10,000 executions performed for each replacement policy. For the MBPTA compliant replacement policies, we also show the average of the 1% of simulations that had the highest execution times. The worst 1% runs for RP and NMRURP perform as good as the average of all runs for LRU and the other deterministic policies. As shown, for the highest 1% execution times, the gap between RR and RP/NMRURP increases due to the bounded pathological evictions with RP/NMRURP. Also, there is a significant gap with LRU, which also triggers some pathological cases in some sets for some placements, thus showing that RP and NMRURP are the best MBPTA compliant replacement policies. LRU only performs slightly better than RR but worse than RP/NMRURP. This occurs because it preserves temporal locality better than RR, but its pathological cases make it worse than RP/NMRURP. Note that L2 miss rates are lower for the worst 1% than on average for all runs. This relates to the fact that DL1 accesses dominate execution time, and higher execution times occur for higher DL1 miss rates. Thus, although the number of L2 misses remains barely constant for the worst 1%, the number of accesses increases and hence, the L2 miss rate decreases.

Differences between RP and NMRURP in terms of average performance are marginal for the evaluated benchmarks. While differences among them exist and may be relevant in some specific cases, most programs do not exhibit often those specific cases where NMRURP is superior to RP and hence, their average performance is roughly identical. Although NMRU and BT are not compatible with MBPTA, as discussed before, we also include them in this evaluation, showing that their performance, both across all measurements and across the worst 1%, is roughly identical to that of RP and NMRURP, which, however, attain MBPTA compliance.

Table 5.2: Replacement policies: average results for the 10,000 executions and the worst 100 (1%) for the railway case study

	All					
	LRU	**NMRU**	**BT**	**RR**	**RP**	**NMRURP**
Cycles	3299	3288	3288	3311	3288	3289
IL1 m.r.	0.151	0.151	0.151	0.152	0.151	0.151
DL1 m.r.	0.302	0.302	0.302	0.305	0.302	0.302
L2 m.r.	0.781	0.781	0.781	0.776	0.781	0.781
	Worst 1%					
Cycles	3389	3306	3299	3408	3305	3306
IL1 m.r.	0.151	0.151	0.151	0.153	0.151	0.151
DL1 m.r.	0.303	0.314	0.307	0.312	0.315	0.314
L2 m.r.	0.794	0.772	0.776	0.783	0.772	0.772

For the sake of completeness, we have also considered 8-way caches, despite the target processor (LEON4 multicore) only implements 4-way caches. However, other embedded processors also implement 8-way L1 caches. Since the L1D cache has shown to be the most sensitive cache to the replacement policy used for this benchmark suite, we have only varied L1D cache set-associativity (using 8 instead of 4 ways). Results across all benchmarks barely changed, showing less than 1% variation w.r.t. the 4-way setup in terms of execution time. For instance, results for NMRURP with an 8-way DL1 are 44789 cycles on average, and 45071 cycles for the worst 1%. Since no further insight was observed, detailed results have been omitted.

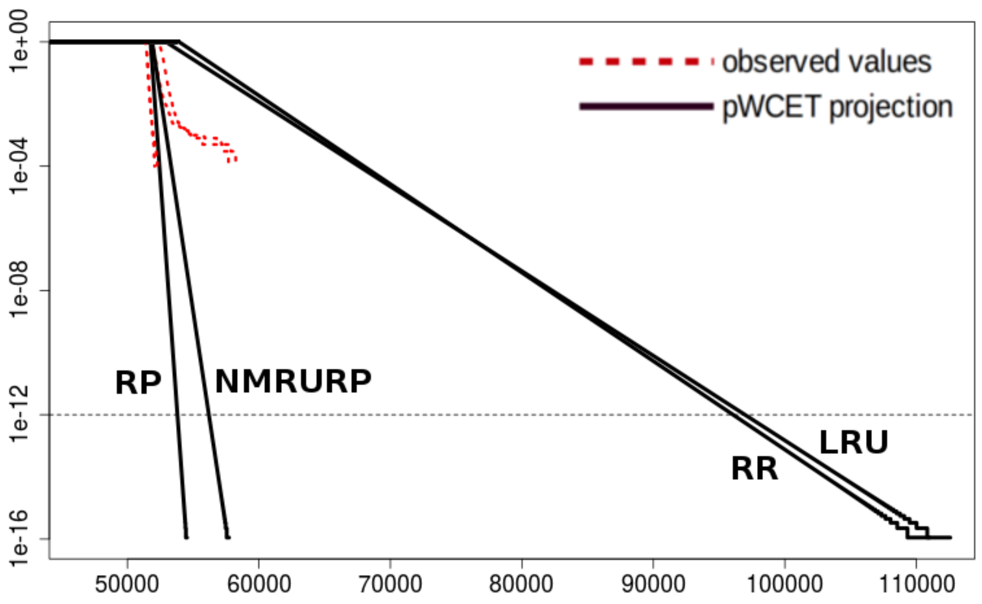


Figure 5.12: pWCET for `jfdc-tint` Mälardalen Benchmark for all MBPTA compliant replacement policies

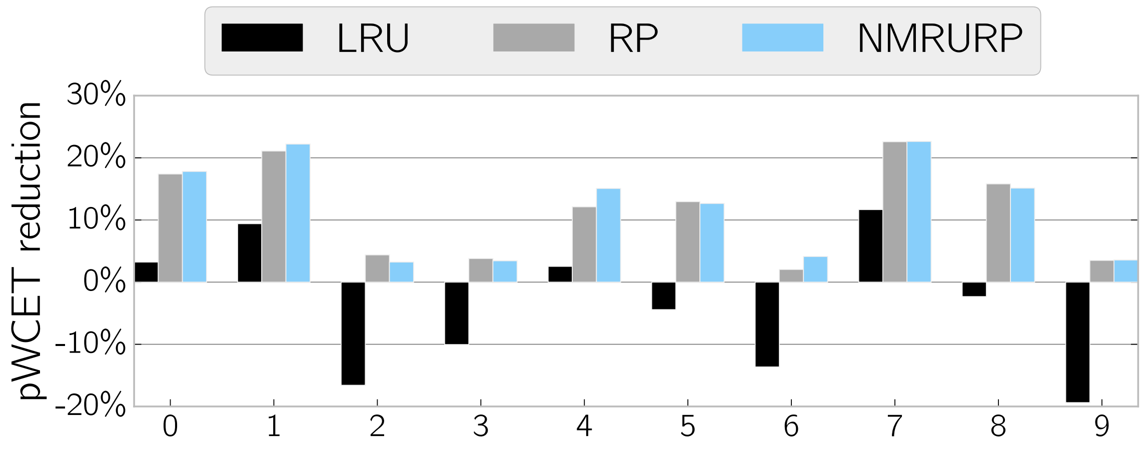


Figure 5.13: pWCET reduction ($p = 10^{-12}$) for the rail case study for all MBPTA compliant replacement policies

5.5.2.3 Railway case study

For the railway case study, average results across input sets are shown in Table 5.2. As shown, RP and NMRURP provide small gains w.r.t. RR in terms of cycles, DL1 and IL1 miss rates. RP and NMRURP are slightly worse in terms of L2 miss rate. When compared against LRU and the other deterministic policies, we also observe negligible differences. However, if we keep only the highest 100 measurements (the worst 1% of them), we observe that differences increase, evidencing that RR may produce pathological scenarios *ps1* and *ps2*, as we have further verified inspecting the sequences of events for the worst RR runs. Conversely, RP and NMRURP limit the maximum number of evictions so that its worst case is better than that of RR. As for Mälardalen, LRU performs slightly better than RR but worse than RP/NMRURP since LRU produces sporadic but significant pathological cases.

Analogously to the case of Mälardalen benchmarks, differences between RP and NMRURP in terms of average performance for the railway case study are marginal, and NMRU and BT, which are not MBPTA compliant, perform roughly as RP and NMRURP.

5.5.3 Worst-Case Performance

As shown in the previous section, the differences between RR/LRU and RP/NMRURP grow at the tail of the distribution (i.e. for the highest values): for instance, average execution time for RP is 1% lower than for RR and 3% lower for the 1% highest execution times of RR. The latter translates into tighter pWCET estimates for RP/NMRURP.

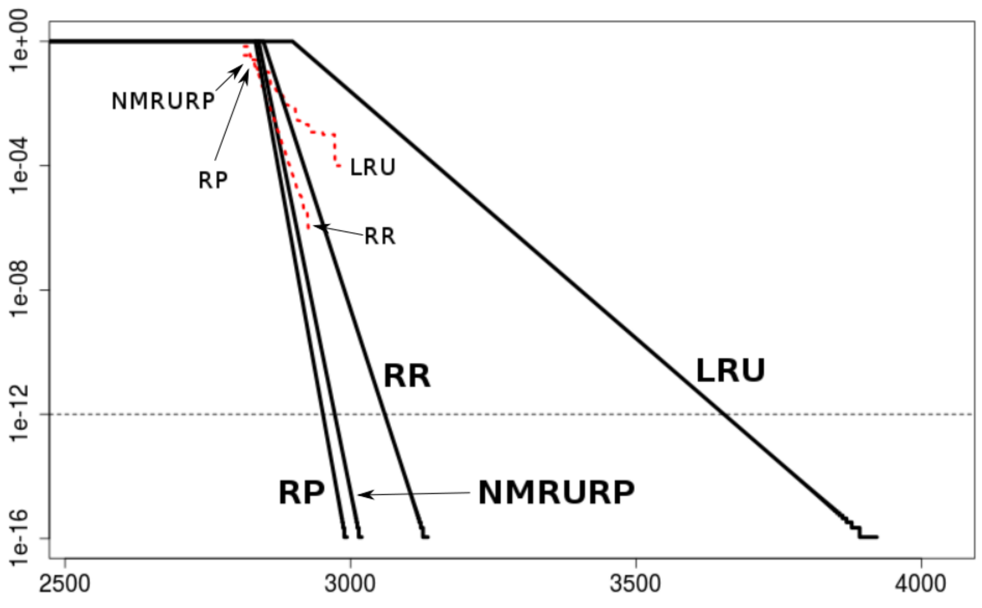


Figure 5.14: Replacement policies: pWCET curve for input 9 of the railway case study w.r.t. RR

5.5.3.1 Mälardalen

For Mälardalen, in Figure 5.11 we present the reduction of pWCET estimates at 10^{-12} w.r.t. those of RR[2]. We observe that RP and NMRURP are consistently better than those for RR. The improvement ranges from 1% to 86%, being 24% on average for RP and NMRURP. This significant reduction in pWCET evidences the advantage of using RP or NMRURP instead of RR or LRU. LRU is on average 1% better than RR, performing a better on some cases and worse in others.

While discrepancies between RP and NMRURP are larger in terms of pWCET estimates than in terms of average performance, they are low across individual benchmarks and marginal on average. In fact, if we consider confidence intervals of 95% for pWCET estimates, intervals overlap for RP and NMRURP, thus indicating that differences are not statistically significant. Part of our future work consists of investigating whether specific patterns causing different behavior between RP and NMRURP exist to a sufficient extent in some industrial programs so that a better selection among RP and NMRURP replacement policies can be performed.

Figure 5.12 shows the pWCET distribution when using RR, LRU, RP and NMRURP for the `jfdctint` Mälardalen Benchmark. Red dotted lines and black straight lines represent the CCDF for the measured data and the pWCET curves respectively. RP and NMRURP provide increasingly higher gains as the exceedance threshold decreases due to the fact that RP and NMRURP avoid pathological evictions by construction. Since RR can produce some such pathological evictions with relevant probability, MBPTA accounts for that by smoothening the shape of the curve and shifting it to the right. Analogously, LRU can produce some pathological cases, which has also some significant impact on pWCET estimates.

5.5.3.2 Railway case study

For the rail application Figure 5.13 shows the pWCET estimates at 10^{-12} w.r.t. those of RR. We observe that the pWCET estimates of RP and NMRURP are consistently better than those for RR , while LRU is sometimes better and sometimes worse, although on average LRU performs worse since its pathological cases can occur systematically as opposed to those of RR, which occur with decreasing probabilities. RP pWCET reduction w.r.t. RR is 11% on average, reaching 22.6% for input set 7.

For illustration purposes, Figures 5.14 shows the pWCET distribution when using LRU, RR, NMRURP and RP for the railway case study (input set 9). Observed trends are similar to for Mälardalen benchmarks: the lower the exceedance probability considered, the larger the gap between RP/NMRURP and RR/LRU. Moreover, in this particular case, we observe that LRU is significantly worse than RR (almost 20% worse) due to the systematic nature of its pathological cases.

Overall, RP and NMRURP provide slightly better average performance than RR,

[2]BT and NMRU have been excluded from this comparison since they are not MBPTA compliant and hence, the result of applying MBPTA on those policies would not have a practical meaning. Thus, despite the values obtained are very similar to those of RP and NMRURP, they are not true pWCET estimates and cannot be used in the context of CRTES.

and similar performance as deterministic policies. However, in terms of pWCET estimates, our proposed replacement policies, RP and NMRURP, are consistently better than RR and LRU by preserving locality and avoiding pathological cases by construction.

5.6 Related Work

Research on replacement policies is abundant, but often targets either improving average performance or achieving deterministic predictability. Among those we find FIFO and LRU replacement policies as well as enhanced versions of them such as protected LRU [85] and pseudo-LRU, which has already been deployed in some IBM processors [145]. A performance comparison of these replacement policies including also NMRU is presented in [10]. Also, an oracle replacement policy has been defined as a reference but not made implementable [20]. Work on optimizing replacement policies for second (L2) and third level (L3) caches is abundant [129, 41, 79, 19, 135]. Those works, either for uniform [129, 41, 79] or non-uniform [19, 135] cache access architectures, leverage the fact that L1 caches filter many accesses, so that access patterns in L2 and L3 caches differ noticeably from those in L1 caches. In general, those cache policies have systematic pathological cases due to their deterministic nature, thus being unfriendly for MBPTA, as it is the case for LRU.

5.7 Conclusions

In this work we analyze the impact of cache replacement policies showing that deterministic ones can cause systematic pathological cases, thus degrading the quality of the WCET estimates. Conversely, RR makes pathological cases non systematic, but they can still occur with decreasing probabilities. This ultimately enforces MBPTA to account for some unfortunate cases with large number of random replacements.

We, then, propose two new randomized replacement policies, RP and NMRURP. We show that they completely remove pathological cases by preserving cache locality to some extent. This allows avoiding pathological cases and hence, improving WCET estimates drastically. Our evaluation on a set of benchmarks and a railway case study show that both policies largely outperform RR in terms of WCET estimates, despite average performance gains are rather modest. Whether one of the two randomized replacement policies proposed in this Chapter, namely RP and NMRURP, is superior to the other remains to be proven since differences among them in our evaluation cannot be regarded as statistically significant. Thus, as part of our future work we plan to verify whether large discrepancies among both policies can be found in other applications.

Chapter 6

Cache write policies

6.1 Introduction

The cache write policy determines how writes to lower (L1) cache levels, those closer to the cores, are handled. Under write-through (WT), write operations are performed in the lower cache and are forwarded to the higher (L2) cache level so that both caches hold consistent data. With write back (WB), write operations are only performed in the lower level cache, and the update of the next level is postponed until the cache lines containing the dirty data is evicted from the lower level cache. The write policy impacts the write-miss policy (write-allocate or not write-allocate), the cache coherence solution (e.g. in snooping-based protocols the write miss policy determines – together with the inclusivity protocol – the set of actions to take on a read/write to local and global data), and the reliability solution (e.g. WT usually requires low-overhead parity in lower level caches and ECC in higher level caches, whereas WB requires ECC in dL1 to keep the reliability of data not backed up in L2). Due its remarkable impact on the overall MLC cache design, the write policy affects metrics as important as guaranteed performance, energy/power, and reliability.

Interestingly, each write policy offers a different trade-off among the different metrics and Multi-Level Caches (MLC) complexity. Hence, the design of the write policy requires finding a balance between them. The latter goes beyond a simple high-performance and real-time classification. Instead, for a given area (e.g. real-time), the particular application domain defines the relevance of each metric and hence, the write policy to use. For instance, in the space domain, due to exposure to radiation, hardware reliability plays a much more important role than in railway. Likewise, performance is much more relevant in automotive, where performance needs are expected to increase by 100x in coming years [2], than in space. In this line, we make the following main contributions:

1. We make an in-depth analysis of both write policies, WT and WB, with emphasis on those metrics of relevance for real-time systems. WT simplifies coherence since most updated data is always in L2, and reliability since the more costly ECC is only needed in L2 with only parity being used in dL1. However, as the pressure on the interconnection (NoC) increases – as a result of integrat-

ing more cores – the contention on the NoC generated by writes under WT greatly reduces guaranteed performance (i.e. increases Worst-Case Execution Time (WCET) estimates). Further, WT increases energy consumption as each write accesses the NoC and the larger L2. With WB, each write to dL1 does not result in accessing the NoC, with considerable energy consumption reduction; and exceptional WCET reductions. Yet, WB complicates coherence and reliability, increasing cache complexity.

2. We propose Hybrid Write Policy (HWP), a low-overhead mechanism that takes advantage of the good properties of each policy. Building on WT, we attack its average and guaranteed performance issues, with a mechanism that builds on shared/private data classification hardware and applies WT to shared data and WB to private data. HWP removes write-through operations on private data, which in general are the most accessed data, while keeping it for shared data, so cache coherence can be managed as in pure WT caches. At hardware level, in the Memory Management Unit (MMU) or Memory Protection Unit (MPU), HWP incurs negligible cost for tracking whether memory pages are shared or private along with other page properties such as read/write permissions.

3. We evaluate WCET estimation, reliability, energy consumption and coherence cost of HWP. Our results show that for those scenarios in which tasks have limited data sharing, HWP delivers performance similar to WB. Even when the percentage of shared data is as high as 40% HWP remains competitive in all evaluated metrics (other works estimate the percentage of shared data in multiprocessor programs ranges from 25% [67] to 17% [73]). Overall, our design has a simplicity comparable to WT in terms of coherence, while achieving average/guaranteed performance and energy consumption comparable to WB.

6.2 Tradeoffs in the Design of Cache Write Policy

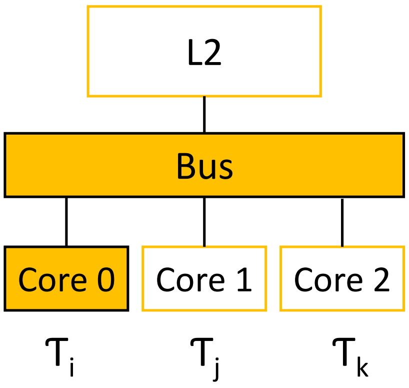
MLC are one of the main hardware blocks in a multicore architecture devoted to improve performance and reduce the energy/power profile of applications. MLC aim at rapidly and efficiently satisfying data/instruction requests coming from the cores, while maintaining the coherence (i.e. the particular value returned on a read), consistency (i.e. when data is available), reliability (physical integrity) and more recently security (i.e. protection against unwanted/unauthorized actions). The cache write policy, which handles write operations, is at the core of the complexity of MLC since it has a direct impact on the design of other policies. In this section we analyze the impact of WT and WB policies on reliability, inclusivity, and coherence choices. We also analyze their impact on performance (average and guaranteed), reliability, and energy/power. For the latter, the results obtained from several controlled experiments are used as supporting argument.

Table 6.1: Percentage of stores executed by the EEMBC Automotive and MediaBench suites

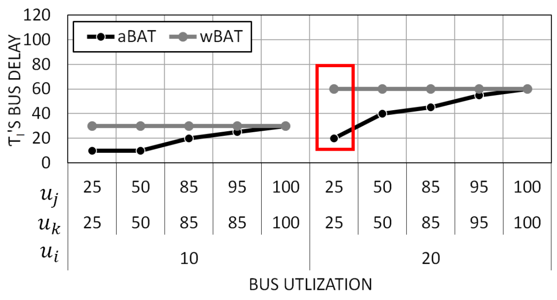
EEMBC	%	MediaBench	%
a2time	5%	adpcm.d	13%
aifftr	18%	adpcm.e	14%
aifirf	8%	epic.d	6%
aiifft	18%	epi.e	5%
basefp	2%	g721.d	8%
bitmnp	11%	g721.e	9%
cacheb	16%	gsm.d	3%
canrdr	15%	gsm.e	3%
idctrn	8%	jpeg.d	6%
iirflt	7%	jpeg.e	10%
matrix	3%	mesa.m	12%
pntrch	0%	mesa.o	14%
puwmod	12%	mesa.t	9%
rspeed	14%	mpeg2.d	10%
tblook	6%	pegwit.d	6%
ttsprk	4%	pegwit.e	6%
		pgp.d	5%
		pgp.e	13%
		rasta	8%

6.2.1 Write-Through (WT)

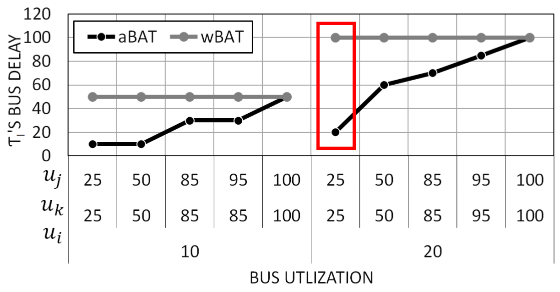
Under WT, each store operation is sent to the L2 so it uses the NoC, which can significantly increase the pressure on it. In the core, the store buffer decouples the commit (finalization) of the stores so that they do not block the pipeline. To that end, once a store reaches the commit/writeback stage, it updates dL1 and in parallel it is placed in the FIFO store buffer allowing the execution to continue. The store is forwarded to L2 when it reaches the head of the store buffer and there is available NoC bandwidth. The store buffer can significantly mitigate the impact of stores in single-core architectures, but rapidly becomes insufficient in multicore. This is better illustrated with an example: let us assume that a bus connects dL1 and L2 caches and each store operation uses it for k cycles. As long as the frequency of stores is (on average) below $1/k$, they will not significantly affect processor performance – unless they are bursted which we do not assume in this simple example. However, in a multicore architecture with N_c cores, as soon as the pressure in the bus increases, the actual duration of a store becomes $k \times N_c$, i.e. $k \times (N_c - 1)$ cycles of contention and k cycles for the bus access. In this scenario, stores become a performance issue as soon as their density reaches $1/(k \times N_c)$. As an illustrative example, Table 6.1 shows the percentage of store operations executed by EEMBC and Mediabench benchmarks, see Section 6.4 for more details on the experimental setup. The average percentage



(a) Setup



(b) Same duration ($l^i = l^j = l^k$)



(c) Different duration ($2 \cdot l^i = l^j = l^k$)

Figure 6.1: Contending tasks in a multi-core: τ_i aBAT and wBAT as a function of its load and its contenders' (τ_j and τ_k) load

of stores is 9%. Further $k \times N_c \in [15, .., 20]$ – and hence it is higher than 1/9, for multicores with 4-8 cores. To make things worse, the percentage of memory operations is growing in emerging data-intensive real-time applications, e.g. applications in cars managing data coming from different sensors such as radar, LIDAR, and stereo cameras. Intuitively, this problem can be alleviated by using a crossbar between the dL1 and the L2, at the expense of increased hardware cost. However, this would just shift the problem from the bus to the L2 itself, since L2 access latency is longer than that of the crossbar. Further, to preserve coherence, each store must be allowed to reach any part of the entire L2 cache, which defeats any attempt to mitigate the problem by partitioning the cache space.

The impact of WT on average performance due to NoC contention magnifies for guaranteed performance, causing inflated WCET estimates. This comes from the fact that worst-case time allowances must be done in the WCET estimates to factor in the impact of NoC contention. In general, no assumption can be made on how the requests of the different running tasks are interleaved in the use of the bus. The exception to this are some Static Timing Analysis (STA) techniques that keep track of the worst-time when each request from each core can be issued, and hence are able to exactly determine how requests overlap in the access to shared resources [104, 68, 103]. This, of course, comes at a significant cost, including the increasing effort of making a cycle-accurate model of the MLC system and processor, and increased analysis computation time. Further, this analysis, despite producing (in general) tighter WCET estimates, makes them non time-composable so that any shift in any task requires performing the WCET estimation for all tasks. Hence, to increase time composability and reduce costs, worst-case assumptions are made on how tasks' request are aligned [117, 51, 87]. This is better illustrated with an example. Let us assume a bus connecting the L2 to 3 cores (respectively executing tasks τ_i, τ_j, τ_k) and all bus requests using the bus for the same duration l (shown in Figure 6.1 (a)). The best overlapping scenario for average Bus Access Time (aBAT) happens when requests of the task under analysis (τ_i) and the contender tasks (τ_j, τ_k, ...) do not overlap as long as the bus utilization of all tasks is below 100%, and when the utilization goes over 100% the minimum overlap happens. For instance if a τ_i uses the bus for 20% of the time and τ_j for 90% of the time, τ_i gets affected only 10% of its time. The worst overlapping scenario for bus access time (wBAT) is assuming that requests from τ_i arrive in the same cycle as the requests from the contender tasks, but τ_i systematically gets the lowest priority. Figure 6.1(b) shows how worst-case BAT gets much more affected than average BAT due to contention for different scenarios of bus utilization of τ_i and its contenders. We see that wBAT is significantly affected even for low bus utilization. For instance, for utilization $u_i = 20\%$, $u_j = 25\%$, $u_k = 25\%$ for τ_i, τ_j, and τ_k respectively (see red rectangle in Figure 6.1(b)), τ_i suffers no delay in the best case aBAT and in the worst case it goes to 60% (a 2.4 increase). Further, typically store operations take shorter than load operations accessing the cache (no need to wait for a response), which translates into a scenario in which τ_i requests take shorter than its contenders' request. We see in Figure 6.1(c), for a scenario in which τ_i requests take half of its contenders, that the impact of contention on WCET estimates increase. For instance, for utilization $u_i = 20\%$, $u_j = 25\%$, $u_k = 25\%$ for τ_i, τ_j, and τ_k respectively (see red rectangle in Figure 6.1(c)), τ_i suffers no delay in aBAT but a 100% in the wBAT.

Continuous store accesses to the L2 cause performance and WCET degradation but can also increase power consumption. Updating values with WT policy implies accessing the bus and L2, even if the core updating the values is the only consumer of this data. This can have a significant impact on the overall power consumption. For example, when running `a2time` from the EEMBC automotive benchmark suite in our reference processor setup (see Figure 6.4 and Section 6.4.1), the 14% of the energy consumption comes from the bus and L2.

Under WT, reliability can be handled with reduced overhead. A usual tradeoff consists of using only parity for error detection in dL1 caches, and (usually) apply it at double-word level, that is, using 1 parity bit for 64 data bits (8 bytes). This results in low overhead of around 1.6% (1/64). Furthermore, the operations needed to compute the parity (XOR) can be carried out in parallel and hence are unlikely to affect cycle time. On a parity error, however, hardware support is needed to squash the execution of the instruction that obtained erroneous data and following instructions. On completion, error-free data is fetched from L2 and execution resumes. Alternatively, parity can be checked before delivering the data to remove the need of squash logic. However, this would likely increase cache latency since XOR gates to compute the parity bit may easily need an extra cycle. WT parity-protected dL1 caches are used in combination with SECDED-protected L2 caches. The latter is achieved with ECC that carries an inherent area and logic for its implementation.

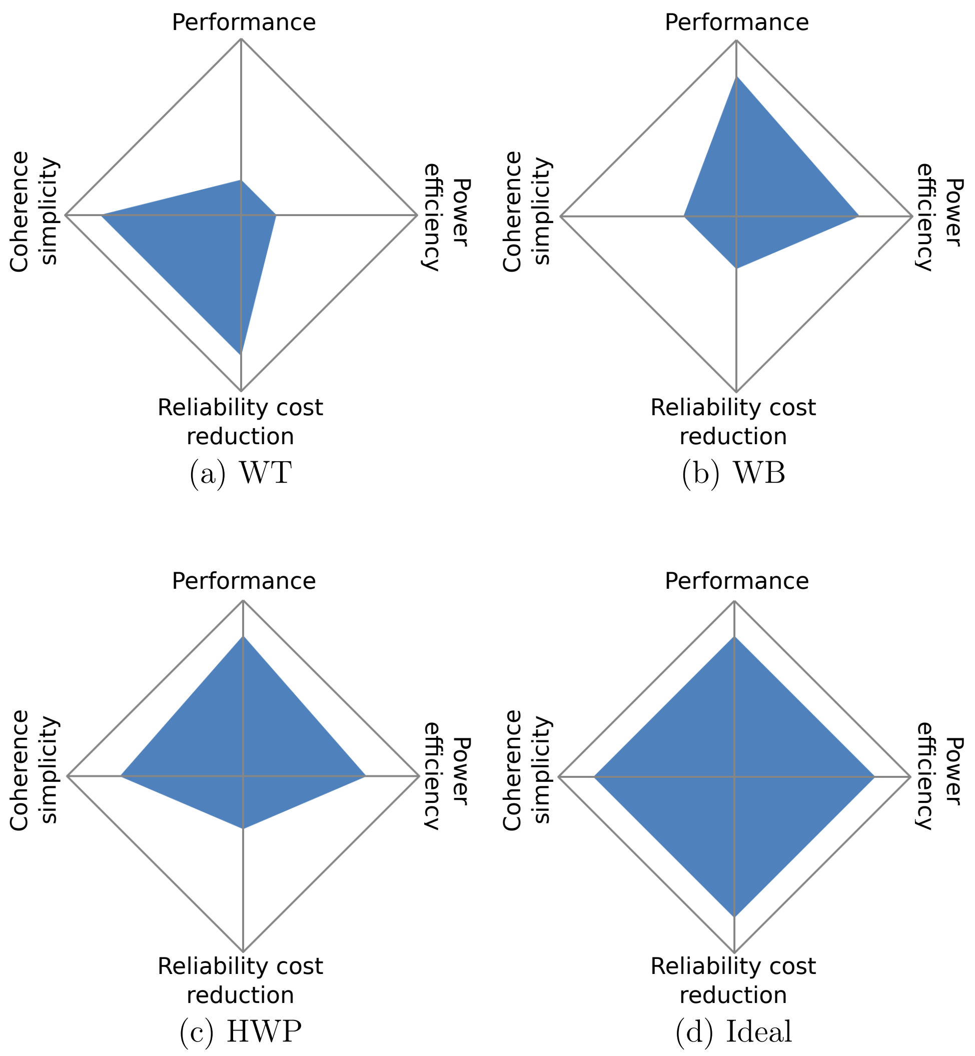


Figure 6.2: Visual comparison of the different write policies for different metrics

Typically, SECDED requires 8 code bits per 64 data bits (so $\approx$ 12.5% extra bits), with negligible impact on L2 performance, since although an additional cycle may be needed to deliver data corrected, this operation is fully-pipelined. Hence, L2 latency may increase by 1 cycle, thus slightly increasing the latency of dL1 read misses, which are generally scarce, but without affecting L2 throughput. Note that on the event of detection of an error in dL1 in a given cache, it is simply discarded and data is fetched from upper cache levels since a correct copy of the data exists in L2 or beyond.

dL1 WT caches simplify coherence management. In particular, dL1 WT caches are made inclusive L2. As a result, when shared data exists in data dL1 (dL1), up-to-date copies of the data is also present in L2. Hence, coherence can be managed in L2 and, upon shared data modifications, the corresponding cores' dL1 caches receive (infrequent) invalidation requests. With WT caches a simple invalidation protocol (V/I)) is enough.

Overall, WT can negatively affect average and worst performance – the latter more intensely– and energy. On the positive side, it can be used with low-overhead coherence and reliability solutions. These properties are summarized in Figure 6.2(a) in a qualitative manner, with Figure 6.2(d) showing the ideal scenario.

6.2.2 Write-Back (WB)

For low core counts, the small average performance improvement of WB over WT does not compensate its additional validation and design costs. However, as the number of cores of multicore real-time systems increases, WB becomes more attractive.

WB significantly reduces the number of bus and L2 accesses compared to WT. Furthermore, since worst-case contention is quite proportional to the number of accesses, WCET estimates are typically much lower with WB than with WT.

WB access count reduction to shared resources decreases the power consumed by those resources. In our setup, the bus and L2 accounts on average for 13% of the system energy, and hence reducing its utilization translates into a non-negligible energy reduction. Also, the need for higher reliability in the data dL1 cache (dL1) increases the power used by the system due to the extra bits and logic needed to implement, for instance, SECDED codes. Finally, since invalidation operations due to shared data accesses may require invalidating dirty lines in dL1, this may cause extra energy consumption to write data back to L2.

When WB is used in the dL1, the data most frequently updated/sensible can be spread between multiple caches (the different dL1 caches and L2). In this scenario, error detection in dL1 and error correction in L2 is not enough, since some data is only updated in dL1 and, upon an error, it could be detected but not corrected. In this case, there are two possible implementations of ECC in dL1, each one with its advantages and drawbacks:

- Under **Data delivery after correction** data is read from dL1, then ECC checked (and eventually data corrected), and finally data is delivered. Unfortunately, checking the ECC code increases access latency by 1 cycle. While such operation can be pipelined, thus not increasing dL1 utilization, the effective latency for data read increases.
- Under **Data delivery before correction** data is read from dL1 and delivered as if it was error-free. In parallel, ECC is checked and, upon an error detection, the affected instruction and subsequent ones need to be squashed. Then, the execution can be resumed using the corrected data. While such process has negligible impact in performance (radiation errors occur only sporadically), the logic for squashing instructions and resuming execution may be complex. However, such logic is analogous to that of WT caches when operating with parity.

With WB caches V/I is not enough because data can be in another state apart from valid or invalid, namely, modified state. Because of this, we will use MESI (an enhancement over MSI) for WB caches. Maintaining cache coherence in multicores with WB dL1 caches requires frequent accesses to other cores' dL1 caches to verify whether shared data is there and, eventually, retrieve them (if dirty) or invalidate

Table 6.2: Commercial processors and their relevant write policy and general characteristics

Processor	**Cores**	**Frequency**	**L1 WT?**	**L1 WB?**
ARM Cortex R5	1-2	160MHz	Yes, ECC/parity	Yes, ECC/parity
ARM Cortex M7	1-2	200MHz	Yes, ECC	Yes, ECC
Freescale PowerQUICC	1	250MHz	Yes, ECC	Yes, parity
Freescale P4080	8	1.5GHz	No	Yes, ECC
Cobham LEON 3	2	100MHz	Yes, parity	No
Cobham LEON 4	4	150MHz	Yes, parity	No

them (if the ongoing access is a write or data is not dirty). Depending on the inclusivity of the cache system, we find two possible scenarios (exclusive caches are infrequent so we do not discuss them here):

- **Inclusive**. In an inclusive cache system (the most convenient solution) the updated data is in dL1 or L2, but L2 has all the tags. This means that all coherence requests can go to L2, and only upon a match ask dL1 for the data it needs.
- **Non-inclusive**. If the system is non-inclusive, there is no unique cache that "knows" where all the data is. This means that any request for data has to be communicated to all caches (all the private dL1 and L2), and any cache can answer with the data. This complicates the coherence protocol design. Hence, we disregard this option.

Either case, whenever some data is requested and the L2 experiences a hit on shared data, it must stall the request and block further L2 accesses. Then, the corresponding dL1 caches deliver the data if dirty. Since dirtiness in dL1 caches is not known a priori by the L2 cache, it must remain blocked long enough to allow the dirty data to be read from the corresponding dL1 and be sent to L2. Then, the L2 can update its contents, deliver the data and hence, serve the request. However, the complexity of the logic to manage all this process synchronously and across multiple cycles and components may affect critical circuit paths, which can carry a reduction of the operating frequency.

Figure 6.2(b) presents in a graphical manner the assessment we have done on WB. We can see that while WT is better in reliability and coherence simplicity, it performs worse on performance (both average and worst-case) and power.

6.2.3 Cache Write Policy in Some Commercial Architectures

To better illustrate the quandary chip vendors face when selecting the write policy, we have analyzed the miss policy of several commercial processors[1].

[1]Core and frequency numbers have been obtained from specific processor implementations [148, 142, 118].

The ARM Cortex R5 [16] is a 1 (or 2) core processor that implements both WB and WT in the dL1 cache, both with parity and ECC. This means that either policy can be selected in a configuration register. The ARM Cortex M7 [15] is a low-performance processor. Like the previous one, it implements both write policies in the dL1 cache, but it only has ECC in the L2 cache. ARM acknowledges that using dL1 ECC may have an impact on operating frequency due to the XOR trees for the ECC when getting the data from the cache. Thus, depending on the particular chip implementation of the ARM IP processor we might have to decrease maximum operating frequency or require two cycles to access the dL1 to support ECC in the dL1 and have the possibility of recovering from errors in the cache when WB is enabled. Hence, despite in general WB caches perform better the strong reliability constraints in safety-critical systems and the associated overheads incurred due to implementing ECC in WB caches makes chip vendors offer the users the possibility to choose between WT/WB according to the needs of their application. However, this forces chip vendors to carry with the effort and responsibility to implement and validate both.

The Freescale PowerQUICC [136] implements WB in the dL1 with parity and the L2 with ECC. This lead to a system where not all cache bit-flips can be recovered. In that respect, Freescale states that the probability of errors is so low that the target application domain should accept the possibility of having “unrecoverable" errors.

The Cobham Gaisler LEON3 [60] is a dual-core running at 100MHz, with a 5 stage in-order pipeline. It is designed for critical real-time systems, and implements WT in the dL1 cache, so that reliability can be handled in L2 with more robust ECC. The LEON4 [58] comprises with 4 cores running at 150MHz with a 7 stage pipeline. It has the same critical real-time systems scope as its predecessor, and the same write policies in the dL1.

WT has been widely implemented in the last level of private caches (mainly in dL1) due to its simplicity (no need for reliability and simple coherence) and its acceptable single-core performance. However, in future multi- and many-cores, the increased number of accesses to shared resources will cause a dramatic increase in average execution time and the WCET estimates. WB caches have performance and energy consumption benefits over WT in mid to high core count processors. However, this performance comes at a complexity cost in the coherence protocol mainly and, to a lower extent, in the reliability mechanisms.

6.3 Hybrid Write Policy (HWP)

HWP low-overhead approach, which we propose, addresses WT average and guaranteed performance issues while reducing overheads w.r.t. WB. HWP eliminates the additional cost of coherence for WB caches and, simultaneously, keeps WT operations limited to a small fraction of write operations so that efficiency is close to that of WB caches, see Figure 6.2(c).

In order to reach its goals HWP builds on the following observations. First, cache coherence management with WB caches is costly and complex because cache lines accessed may reside dirty in local dL1 caches. Second, private data is not affected by cache coherence, so conceptually it is irrelevant whether such data is dirty or not in dL1 caches. And third, a significant percentage of memory data is only accessed by one processor (also in parallel applications) and, thus, does not require keeping coherence (e.g. 75% of the access are reported as private in [67] and around 83% in [73]).

From those observations, we design a new policy (HWP) that manages private data as in WB caches and shared data as in WT caches. With HWP, memory contents are classified at page granularity as either shared or private, which has been shown to be a very convenient granularity for private/shared data classification [73, 46]. In particular, as long as a page contains any shared data, it is (pessimistically) classified as shared. Otherwise, it is classified as private. On a write to shared data, HWP writes it through to L2 cache (a la WT). Meanwhile write operations to private pages are not propagated to L2 (a la WB), hence decreasing contention in the access to L2.

Next, we discuss the key characteristics and implementation details of HWP, with emphasis on how to classify data as private or shared (and the appropriate granularity to do so), how to check whether data is shared or private to decide whether to proceed as in a WT or WB cache, how cache coherence needs to be managed, what the reliability implications are, and how contention in the access to L2 is mitigated.

6.3.1 Data Classification

Orthogonally to HWP, a mechanism is needed to classify data as private or shared. Techniques exist to that end, with some of them [73] already integrated on a real hardware platform (LEON3 processor) and Linux, providing evidence of its feasibility. Interestingly, private/share data classification can be performed at different levels (e.g. cache line size).

Private/shared information can be managed at fine granularity (e.g. cache line level). This would allow a much finer classification but at the cost of higher area and energy overheads [73]. Additionally, performing the shared/private classification at page granularity makes it possible using OS functionality to reduce hardware implementation overheads [46].

Ho et al. [73] and Cuesta et al. [46] show that the most convenient granularity to classify data is page level. With this solution, whenever a piece of data is shared between two cores, the whole page in which the data is is marked as shared. Hence, this solution pessimistically assumes that all data in a shared page is shared. As part of that solution, the information on private/share information can be stored in the Memory Protection Unit (MPU) or Memory Management Unit (MMU) for each page along with other information such as whether pages are user-level or supervisor-level, whether they are read/write or read-only, and whether they are cacheable or not. Such information is often cached in the Translation Lookaside Buffer (dTLB) together with address translation. In most processors dTLBs are accessed in parallel with dL1 caches for fast address translation and for verification of the permissions to read/write in specific memory pages. Hence, they can store private/shared information.

In real-time systems an alternative approach to those hardware approaches is

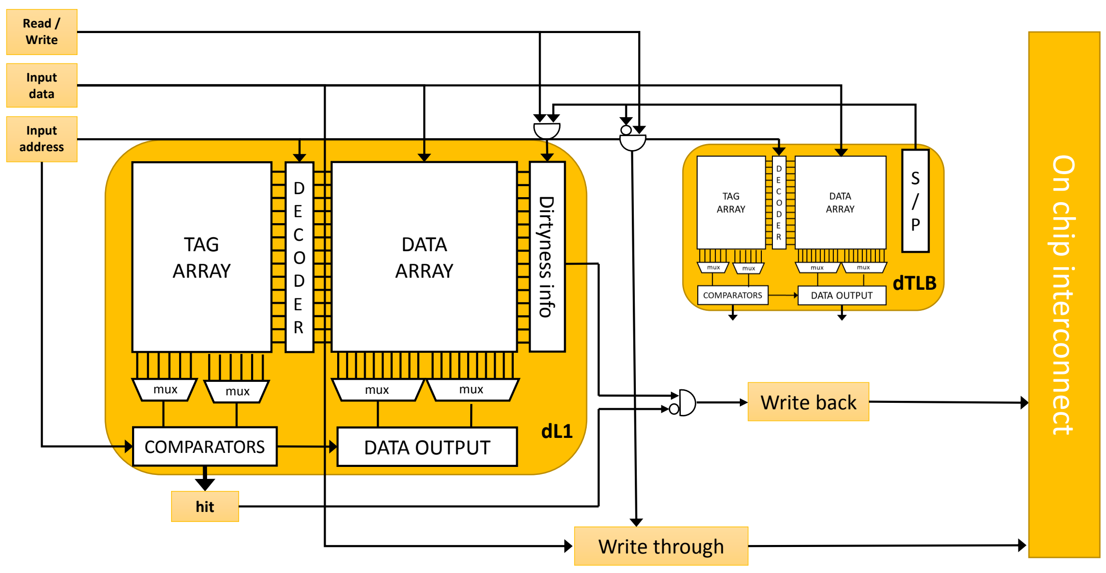


Figure 6.3: Schematic of HWP cache access protocol

possible with software address space partitioning. Many OS use address spaces (i.e. a range of addresses) to map specific I/O devices. Also Real-Time Operating System (RTOS) like PikeOS use separate address spaces to implement resource partition. Furthermore, in the automotive domain, AURIX architectures come equipped with caches and different memory types (e.g. flash, ram). From the software side, address ranges are defined to map data/instructions to the desired memory and or to make data cacheable or non-cacheable. Hence, address space can be partitioned assigning a particular address range to shared data.

The main disadvantage of dynamic hardware solutions is that data re-classification is needed. This happens, for instance, when a page is first loaded by one core (hence classified as private) and then accessed by another core (being reclassified as shared). This does not only create predictability issues in real-time systems, but it also adds complexity to HWP, including writing through all data (dL1 lines) of this page in the owner core, while managed those same data with WB in the other cores. This complexity is avoided with the classification based on software address partitioning, *which is the solution we assume in this Chapter*, without loss of generality.

6.3.2 Private/Shared Data Management

The way in which data is accessed under HWP varies depending on whether data is shared or private. This is graphically illustrated in Figure 6.3.

On a load/store access, the dL1 and the dTLB are accessed in parallel. In case of a dTLB miss, it is served first before proceeding with the access, as done regularly in most processors. Note that address translation is typically needed before accessing the L2. Therefore, while serialization of dTLB misses and dL1 accesses may be unnecessary for some dL1 hits, dTLB miss rates are usually extremely low, and their occurrence together with dL1 hits is even more unlikely since this can only occur if

Table 6.3: Timing of a dL1 hit (dTLB hit) under HWP

L/S		**cycle 1**	**cycle 2**
LOAD		Read dL1, Read dTLB	
STORE	Private	Write dL1, Read dTLB	Update dirtiness bit
STORE	Shared	Write dL1, Read dTLB	Write L2

data from the page has been fetched and sufficient evictions occurred in the dTLB but not in the dL1.

6.3.2.1 Hit in dL1

In case of a dL1 hit (and dTLB hit), the shared/private information determines whether the line hit needs to be marked as dirty or not. If the line belongs to a private page ($S/P = 0$) and the access is a write operation ($W/R = 1$), the dirtiness bit is set. The separation of data and dirtiness information poses no issue since dirtiness information can be accessed systematically one cycle after, as it is only needed in case of a miss to decide whether the evicted cache line needs being written back. Also, in case of dL1/dTLB hit, if the line belongs to a shared page ($S/P = 1$) and the access is a write operation, data is written through L2 as in a regular WT MLC.

In terms of timing, Table 6.3 shows the different possible scenarios and their timing. After the processor request, regardless of whether it is a read or a write, both the dL1 and the dTLB are accessed in parallel. In the case of a read, at the end of the first cycle the data is available and is served to the processor. In the case of a write, at the end of the first cycle the dTLB determines whether it is a write to a shared or private page. If the store targets a private page, the dirtiness bit is updated in the dL1 cache in the second cycle, and the request is completed. However, if it is a write to a shared page, a write request is issued to the L2.

6.3.2.2 Miss in dL1

On a dL1 miss, WT management is performed as for hits. However, if the miss corresponds to a read operation ($W/R = 0$) or the address is private ($S/P = 0$), the line is fetched from L2 and allocated in the dL1 data cache. Note that we assume the usual case where WT implements no-write-allocate (nWA) policy on write misses, whereas WB implements write-allocate (WA). Different write allocate policies could be implemented such as, for instance, WA (or nWA) regardless of the privateness of the data accessed.

In Table 6.4 we see the different scenarios that can happen with a dL1 miss. In the first cycle, both the dL1 and the dTLB are accessed in parallel. At the end of the cycle, if the line to be evicted is dirty, the dL1 sends a dirty eviction request to the upper level. In the load scenario, the next cycle (3 if the line was dirty, 2 if it was clean) the line is requested to the L2. After n cycles, the cache line arrives to the dL1 and the data is be available. In the case of a store private, it also requests the line to the L2. When the answer comes, it updates the dL1 and update the dirty

Table 6.4: Timing of a dL1 miss (dTLB hit) under HWP

L/S		cycle 1	cycle 2	cycle 3	...	cycle n+3
LOAD		Read dL1, Read dTLB	if dirty → Eviction	Request line L2		Read dL1
STORE	Private	Write dL1, Read dTLB	if dirty → Eviction	Request line L2		Write dL1, Update dirty bit
STORE	Shared	Write dL1, Read dTLB	if dirty → Eviction	Write L2		

bit (allocate on private data). Finally, on a store to a shared line, a request for the write is sent to the L2, and no update occurs in dL1 (no allocate for shared data).

6.3.3 Non-Functional Metrics

This section makes a qualitative assessment of the benefits of HWP over WT and WB. Quantitative comparisons are carried out in the Section 6.4.

Under HWP, shared data is consistently stored in L2, making that all shared data in dL1 caches is necessarily non-dirty. As a result, with HWP coherence is managed as in the case of pure WT caches, hence keeping its low- cost and complexity benefits and avoiding the overheads related to WB caches. With HWP V/I is enough, as for WT, because the shared data will always be updated in a single place (L2), so we do not need a Modified state in the dL1 to keep track of who has the most updated data.

Since shared contents are written through to L2, the fraction of dirty dL1 cache contents is smaller than in pure WB caches. Yet some dL1 cache contents can be dirty. Hence, error correction capabilities are still required in dL1, as in the case of pure WB caches. A simple software solution to reduce the associated costs consists in marking the pages storing error-sensitive data as shared. This way the only data that could be lost would be the private one. However, for critical applications, the same reliability technique used in WT (SECDED in dL1) can be used.

Under WT, performance issues relate to contention in the NoC and the L2 due to write-through stores. With HWP, this problem is alleviated, restricting write throughs to stores to shared data. Obviously, the lower the number of accesses to shared data, the lower the number of WT operations, and hence, the lower the contention and the lower the sensitivity to contention. In general, programs are designed to reduce access count to shared data (25% [67] and 17% [73]), which usually carries a serialization of tasks.

In general, power consumption relates to the activity performed (dynamic power) and execution time (static power). By limiting the number of WT operations, dynamic power is reduced drastically w.r.t. pure WT designs. By reducing contention, execution time is also lower than for pure WT designs, thus reducing static power.

Overall, our HWP hybrid cache design offers a globally better tradeoff than WB and WT. This is illustrated in Figure 6.2(c). As shown, our design offers performance and power close to that of WB, with similar reliability overheads, but much lower complexity for the management of shared data. When compared against WT, coher-

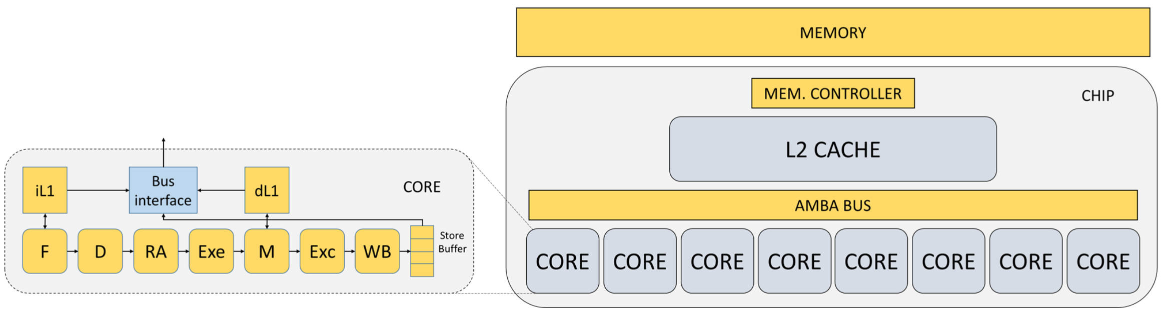


Figure 6.4: Block diagram of the main elements of our NGMP-based 8-core architecture

ence management cost is identical, performance and power are much better, and only reliability costs are higher.

6.4 Evaluation

In this section we quantitatively assess the benefits of HWP over conventional write policies (WT and WB). We use the metrics presented in previous sections, namely, guaranteed and average performance, energy consumption, coherence overhead, and reliability.

6.4.1 Reference Architecture and Benchmarks

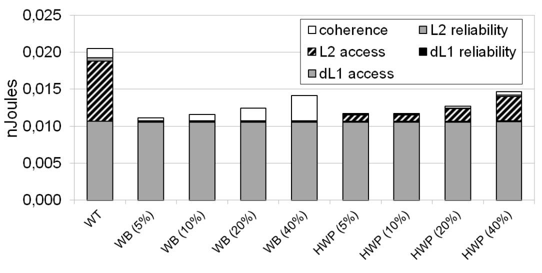
We use the simulator infrastructure described in section 3.3 and the configuration described in Section 3.2.4. As benchmarks we use both EEMBC automotive as well as MediaBench. We create several scenarios in which we vary the percentage of accesses targeting the address range for shared data, as detailed in the corresponding experiments.

6.4.2 Energy

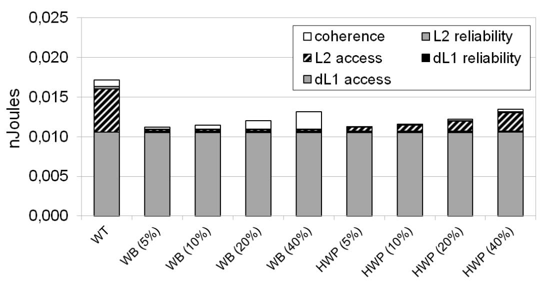
As presented in Section 6.2 each cache write policy carries side effects on the write-miss policy, the reliability solution, inclusivity, and the coherence solution. This affects the set of activities carried out by each task, the energy cost of each activity and hence the overall energy profile of each task. Further, the complexity of each write policy varies which affects its 'intrinsic' energy consumption.

We assess the energy usage under each policy using CACTI [115], the state-of-the-art integrated model for cache and memory access time, cycle time, area, leakage and dynamic power consumption, configured with the NGMP cache parameters. With CACTI we breakdown the energy usage of each cache access into 5 components: dL1 access, dL1 reliability, L2 access, L2 reliability, and coherence.

We present the average cache access energy consumption, across all EEMBC and Mediabench benchmark suites, in Figures 6.5 (a) and 6.5 (b) respectively. The difference across individual programs in each benchmark are not relevant, and hence are



(a) EEMBC



(b) Mediabench

Figure 6.5: Average energy breakdown per cache access for EEMBC and Mediabench

not shown. We compare WT, WB and HWP; and for the latter two, we assume three different scenarios depending on the percentage of accesses to shared data: 5%, 10%, 20% and 40%. Note that WT results do not depend on the percentage of shared data, since all writes go to L2.

We observe that the dL1 energy usage for an access is roughly the same for all write policies. The difference in the energy of the dL1 reliability solution is small, with WT having the lowest value due to the use of simple parity instead of ECC (used by WB and HWP).

We also observe that the lowest access energy profile is obtained for WB and HWT. In the case of WB, there are few L2 accesses, since stores do not access L2 every time, while the load access rate to the L2 is relatively low. HWP has a higher L2 access rate than WB since it writes shared data directly to the L2.

On the coherence side, WB has an increased amount of coherence-related messages as the shared data increases. Taking into account all components, WB and HWP consume roughly the same energy per access for a given ratio of accesses to shared data. Both show approximately a 42-50% per access energy reduction (depending on the percentage of shared data) with respect to WT. To sum up, when comparing the different write policies on the energy aspect, HWP has the same reduced energy consumption as WB compared to WT (up to 50%), but without the coherence complexity inherent to WB, as presented in Section 6.3.

6.4.3 Guaranteed Performance

WCET estimation is one of the most critical metrics for real-time systems, since it determines the guaranteed performance that the system can deliver. As presented before, WCET estimation is challenged by the use of multicores due to contention delay suffered by tasks.

In order to assess the benefits on WCET estimate reduction of HWP, we have created 1-, 2-, 4- and 8-task workloads, as presented in Table 6.5. Workloads have been generated using benchmarks from the EEMBC automotive suite (eembc1.X, eembc2.X) and from the MediaBench suite (media1.X, media2.X). Across workloads, the first task in each workload, the one for which WCET estimates are produced, comprise at least one benchmark with at most a 5% of stores, and at least one

Table 6.5: Benchmark mixes used to assess WCET estimates under different core counts

Mix	main	cont1	cont2	cont3	cont4	cont5	cont6	cont7
eembc1.1	bitmnp							
eembc1.2	puwmod							
media1.1	g721.d							
media1.2	jpeg.d							
eembc2.1	bitmnp	a2time						
eembc2.2	puwmod	aifftr						
media2.1	g721.d	adpcm.d						
media2.2	jpeg.d	adpcm.e						
eembc4.1	bitmnp	a2time	matrix	rspeed				
eembc4.2	puwmod	aifftr	idctrn	ttsprk				
media4.1	g721.d	adpcm.d	gsm.d	pegwit.d				
media4.2	jpeg.d	adpcm.e	g721.e	pgp.d				
eembc8.1	bitmnp	a2time	matrix	rspeed	tblook	canrdr	aifirf	aifftr
eembc8.2	puwmod	aifftr	idctrn	ttsprk	basefp	cacheb	tblook	ttsprk
media8.1	g721.d	adpcm.d	gsm.d	pegwit.d	g721.d	pegwit.d	gsm.e	pgp.d
media8.2	jpeg.d	adpcm.e	g721.e	pgp.d	adpcm.e	jpeg.d	gsm.e	adpcm.e

benchmark with at least a 13% of stores. The rest of the new benchmarks in the workload are selected randomly.

Modeling multicore contention is a concern for timing validation and verification as witnessed by a notable amount of works on the topic, summarized in [54]. Many measurement-based approaches – the most extended industrial practice – build on the availability of performance monitoring counters (PMCs) [114, 117, 47, 81, 51]. From those we build on [81] since it captures the number of requests each core performs to the shared resources. This results in *partially time composable* WCET estimates, rather than *fully-time composable* ones that result from assuming that every single request of the task under analysis is delayed regardless of the load contenders put on the shared resources.

We illustrate the model [81] with a small example comprising one task under analysis or τ_a and a contender task or τ_b. When τ_b has more requests than τ_a, each request of τ_b is assumed to delay the requests of τ_a. The worst-case contention that τ_b can cause on τ_a, i.e. $\Delta^{cont}_{b \to a}$, is computed according to Equation 6.1, where n^t_b is the number of τ_b requests of type t and lat^t is the latency of that request type. Note that the model makes the worst case assumption of no overlap of requests, so each τ_b's request delays τ_a by its latency, i.e. lat^t.

$$\Delta^{cont}_{b \to a} = \sum_{t \in \mathcal{T}} min(n_a, n^t_b) \times lat^t \tag{6.1}$$

In our case the request types are $\mathcal{T} = \{L2h, L2m, s2h, s2m\}$ corresponding to loads hitting and missing in the L2 cache, and stores hitting and missing in the L2 cache respectively, which can be tracked with existing PMCs [59]. The corresponding latency of each of these event type is [81] (in processor cyles): $lat^{L2h} = 9$, $lat^{L2m} = 7$, $lat^{s2h} = 1$, and $lat^{s2m} = 1$.

Note that it does not matter the type of τ_a requests but just it overall number $n_a = \sum_{t \in \mathcal{T}} n_a^t$. That is, the contention τ_a suffers depends on its total number of requests and the number of requests of each type of its contenders (τ_b in this case). The model factors in the case when τ_b has fewer accesses than τ_a that results in some τ_a requests not being delayed by any request from τ_b. The approach presented in Equation 6.1 for τ_b is followed for all the $Nc - 1$ tasks simultaneously running with τ_a, where Nc is the number of cores. The reader is referred to [81] for more details.

Figure 6.6 shows the WCET estimate obtained for the first task under each cache write policy. WCET estimates are shown as the number of cores varies from 1 to 8. In order to simplify the comparison, all WCET estimates are normalized to the WCET estimate of the first task when run in isolation under WB. We see that in all cases the tightest WCET estimates are obtained with WB. HWP obtains comparable results to those of WB and much better than those for WT. The latter gets rapidly worse as the core count increases. Note that Figures 6.6 (a), (b), (c), and (d) are not directly comparable, since for each figure WCET estimates are normalized to that of WB when the task runs on isolation.

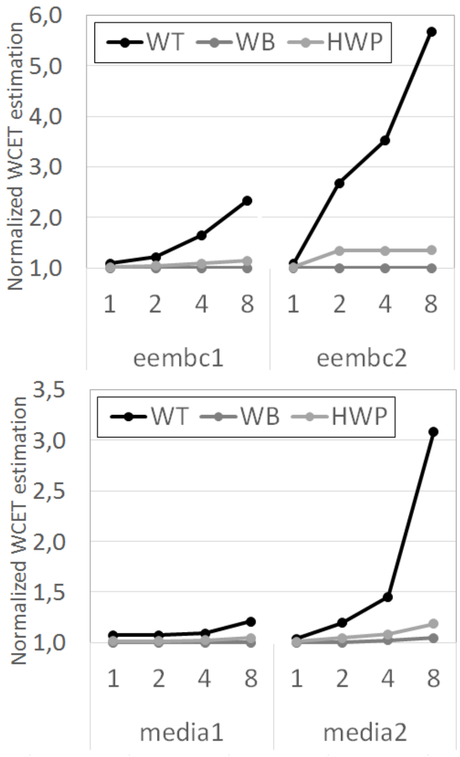
We also see that WT is not affected by the percentage of shared data, since it always updates the L2 regardless if the data is shared or not. WB does not show meaningful variations either, while HWP has small variations (mainly for eembc2 and media2). In all cases HWP is significantly better than WT.

Across all shared-data scenarios for WT we can observe that:

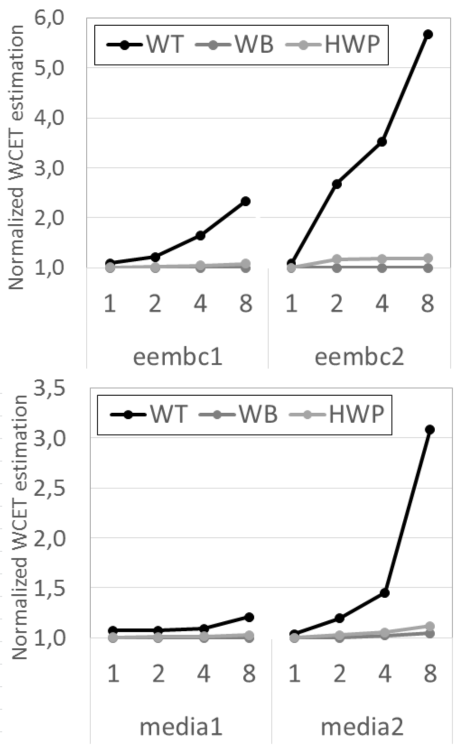
- Mix eembc2 suffers a significant increase in WCET estimates (more than 5x in the 8 core configuration). This is due to the combination of memory instructions the program under analysis executes (30% of all instructions) and the number of stores the competing tasks have (9% of all instructions on average).
- Mixes eembc1 and media2, have lower, yet significant, WCET increases (more than 2x and 3x respectively). This is caused by the combination of the two metrics just mentioned is lower than that of eembc2.
- Finally, media1 has a small WCET estimate increase due to a lower number of memory instructions executed by the main program (23%) and a lower percentage of stores in the challenger tasks (6% on average).

WB is the write policy with lower WCET estimate performance penalty. WB causes a small increase in WCET estimates even when we have 40% of data shared (higher than what is usually found in parallel applications [67, 73]). This is so, because only data requested by other cores is exchange via the bus.

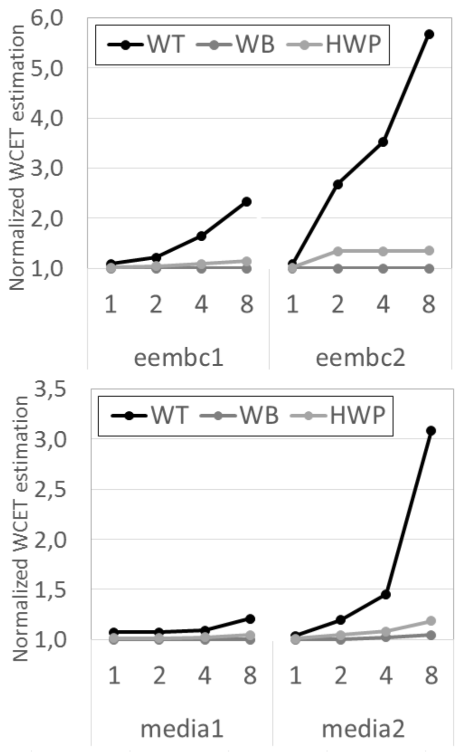
HWP lies in between WT and WB, though it is much closer to WB. For eembc1 and media1, the WCET estimate is remarkably low. This is also contributed by low percentage of memory instructions combined with the low percentage of stores in the challenger tasks. For eembc2 and media2, HWP suffers high increase in WCET estimates in the 40% shared data scenario: despite HWP reduces the pressure on the bus, (i) the high percentage of share data, (ii) the high percentage of memory



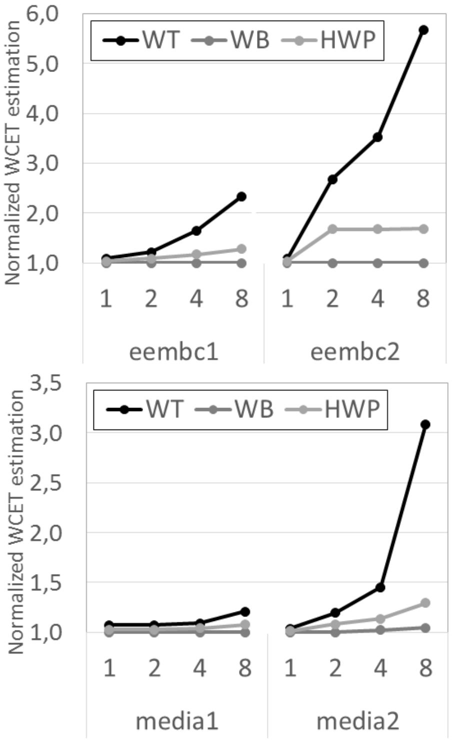
(a) 0% shared



(b) 10% shared

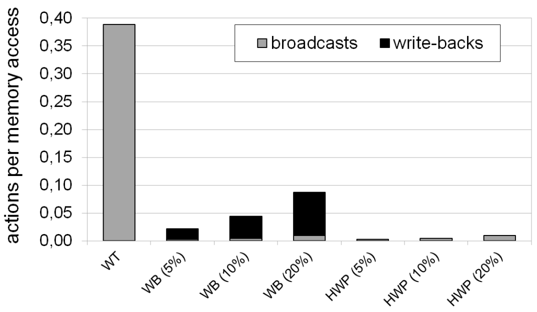


(a) 20% shared

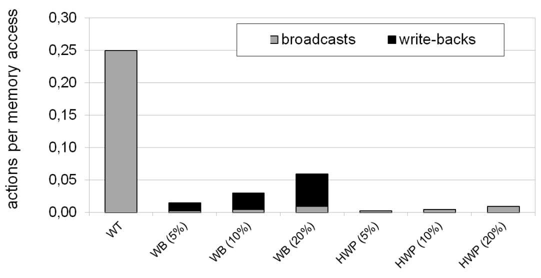


(b) 40% shared

Figure 6.6: Normalized WCET estimate for the first task in the workload under different core counts and percentage of shared data for the different write policies



(a) EEMBC



(b) Mediabench

Figure 6.7: Number of broadcasts and write-backs per memory access

instructions these benchmark mixes execute (30% and 23% respectively), and (iii) the number of instructions that are stores in the competing tasks (8 and 9% respectively), cause the pressure on the bus to increase. Yet, HWP stills performs better than WT, specially in high-core setups (4-8), where WT grows to 3-6x and HWP only grows to less than 2x in the worst setups. The penalty difference with WB as the number of cores and shared data increase is mainly due to the fact that all accesses to shared data is sent to the L2 (write-through on shared data), without the need of another core requesting the data. This means that the same core could write several times directly to L2 without another core requesting the data in between.

To sum up, HWP obtains similar WCET estimates to WB, but significantly smaller than WT (up to 5x) in multi-core setups. This difference in WCET estimates increases significantly with the number of cores being used.

6.4.4 Coherence

The write policy impacts the selection of the coherence solution. With WT caches a simple invalidation protocol V/I is enough, while for WB caches a more complex policy such as MESI is required. For HWP, an invalidation protocol such as the one used in WT is enough.

The potential impact of the coherence protocol, in particular MESI, is two-fold. First, the complexity of its design, implementation, and validation. And second, its impact on performance since the number of messages to exchange between processors and the L2 cache to maintain coherence.

Since the complexity has been qualitatively assessed in Sections 6.2 and 6.3, here we focus on the number of messages that will be sent in every coherence protocol as a proxy to coherence performance overheads. In particular we focus on the invalidation messages and the number of write-backs caused because of coherence (not due to cache capacity issues).

Figure 6.7 shows the average number of coherence messages per memory access for EEMBC (a) and Mediabench (b). We evaluate the 3 write policies: WT, WB and HWP; considering 5%, 10% and 20% of shared accesses in the last two policies. The number of invalidations in WT is high in both benchmark suites because every write access requires that an invalidation message is sent to the bus, since any other private cache can have a copy of the data. For WB and HWP the number of invalidations is much smaller, since the cache directory tracks the core having a copy of each cache line, and only shared data that actually is in private caches will be invalidated.

The other coherence metric we analyze is the number of write-backs related to coherence, which only happen for the WB policy. This occurs when a core c_0 modifies some data in its private dL1 and another core c_1 wants to access that data. Since the L2 knows that c_0 has this data in a Modified state, the L2 asks c_0 to write back the modified data to L2, and then the L2 sends it to c_1. In the WT policy, the L2 always has the most updated values, so there are no write-backs due to coherence. Likewise, HWP only writes back private data, treating shared data like WT, so there are no write-backs due to coherence either.

Note that while the trend for the coherence cost shown in this section is similar to that of energy (Figure 6.5), the absolute values are not the same. This is so because the energy cost of a write-back is higher than that of an invalidation. As a result, when comparing WT to WB/HWP, the energy consumed in coherence is not that high as the number of messages as shown in this section.

Overall, HWP offers the best of WT and WB in terms of coherence: it generates as little invalidations as WB without the coherence related write-backs of WB.

6.4.5 Reliability

We assume that caches are able to detect and correct single-bit upsets (SBU), while multi-bit upsets (MBU) may occur when their probability is high enough. We assume that solutions such as word interleaving[2] are applied so that a N-bit MBU becomes N SBUs. Hence, the criteria to assess reliability consists of whether designs are able to detect and correct single-bit errors. Note that such reliability criteria are already implemented in processors targeting the highest criticality levels in the space [59] and automotive [17] domains.

Since in WT all the updated data is always in L2, only parity is required in dL1 to detect single-bit errors given that correct data can be retrieved from L2. WB and HWP allow dirty data in the dL1 cache, and thus they require error correction capabilities in dL1, such as SEC-DED. The L2 cache always implements SEC-DED, since there can be dirty data at this cache level when using all policies.

The difference in the reliability technique used in dL1 has limited impact on area. Parity, used in WT, imposes a 1.6% increase in the number of cells needed (1 bit per 64-bit word), as well as few XOR gates and a comparator. SEC-DEC, used in WB and HWP, increases by 12.5% the number of cells (8 bits per 64-bit word), and also adds extra XOR gates and comparators [74]. Note that the relative area of dL1 cache w.r.t. L2 cache is typically low, and all write policies implement SEC-DED in L2, thus lowering the relative additional cost of SEC-DED vs parity in dL1 when put in the context of the complete cache system.

[2]Interleaving K words at bit level ensures that bits of a given word, and hence protected with the same parity/ECC code, are at a distance of at least K bits.

This section complements the comparison that has been made in Sections 6.2 and 6.3.3.

6.5 Related Work

Relevant related works relate to WB caches and their use in real-time systems, private/ shared data classification mechanisms, models for computing WCET estimates for multi-core contention, and the use of other hybrid techniques for high-performance computing.

Due to the recent interest in the use of WB caches for critical real-time systems, mainly due to its potential increase in guaranteed performance, some works [140, 31] have studied static WCET analyses of this write policy, since it is more challenging than for WT caches. Authors in [140] propose an eviction-focused technique, analyzing for each cache miss if it could result in a write-back in order to estimate the WCET. In a more recent work [31], a new method has been proposed to complement the previous work by using a store-focused technique. This method consists in checking whether a store may transform a currently clean line into dirty, and hence result in a write-back later on. Those techniques can be retargeted to capture HWP to tighten WCET estimates over WB/WT. In this line, previous works [134] also propose new cache systems that take into account shared/private data to improve WCET estimates, but with more radical changes required in the architecture.

Regarding private/shared data classification, different methods and hardware designs based on them have been proposed [67, 73]. Some authors [67] classify the different types of cache access patterns, and use such classification to implement a specific distributed cache design. Authors study the percentage of data that is private, shared/read-only and shared/modified. In [73] the authors propose a dynamic classification of shared and private pages. This technique needs some WB mechanism when a page changes its status from private to shared. While this technique may also improve performance over WT, it also has to deal with the coherence complexity of WB.

WCET estimation in multicores has been subject to intense study [114, 117, 47, 81, 51]. In [81, 51], the authors propose techniques for computing partially time composable Execution Time Bounds for bus accesses based on the number of requests the contenders can generate, regardless of when they access the bus. These technique provides tighter WCET estimates than simpler fully time composable models that always assume the worst case on a bus access. We have built on these techniques for WCET estimation.

Techniques for a hybrid approach on coherence management have been studied in high-performance domains [130, 46]. In [130], the authors implement a similar technique to [73] that dynamically changes the status of memory pages from private (default) to shared when they are accessed by more than one core. In [46], the authors propose a similar technique to differentiate private and shared pages at OS level, thus reducing the size of cache directories since they do not need to keep track of private lines. In [86] they tackle the task of making a coherence protocol predictible for real-time. However, in these works there is still a non-trivial coherence mechanism with transient states, while our proposal targets a simpler (static) coherence mechanism.

6.6 Conclusions

The relentless trend towards the adoption of multilevel caches in real-time systems is a fact, in the line of high-performance systems. Our analysis of the write miss policy shows that WT simplifies coherence and reliability, while WB performs better in performance and energy. From the analysis we propose a new Hybrid Write Policy (HWP) that discriminates among shared and private data to smartly write through dL1 data or keep it dirty in dL1. Experimental results show that HWP results in remarkably better guaranteed performance than WT. HWP results for energy consumption per memory access improve those of WT. In terms of complexity of the coherence protocol, HWP implements a simple Valid/Invalid protocol like WT, compared to the complex MESI protocol used in WB.

Chapter 7

Cache redundancy

7.1 Introduction

As shown in the previous Chapter, the write policy can have a big impact in guaranteed performance. In particular, a write-through DL1 cache can increase Worst-Case Execution Time (WCET) up to 6x just for the bus contention compared to write-back designs

The sensitivity of caches to errors (faults) is another issue of upmost importance in critical systems. Critical systems must undergo a strict certification process to provide evidence that specific failure rates are below specific thresholds set in applicable safety standards, e.g. ISO26262 [77] in cars. Critical systems include safety mechanisms for fault tolerance to ensure low-enough acceptable failure rates.

It follows that chip designers for critical systems face a conundrum in the design of DL1 caches to provide both reduced WCET estimates (high guaranteed performance) and keep low rates under control. On the one hand, instruction (read-only) caches and write-through DL1 never keep a dirty copy of any data. Hence, they can implement low-cost error detection mechanisms such as parity, since error-free copies of the data exist elsewhere (e.g. in the L2 that is ECC protected). This however comes at the cost of increased WCET estimates. On the other hand, write-back or hybrid write-through/write-back DL1 caches [23] contain the impact of contention in WCET estimates by avoiding that every store access shared resources. DL1 write-back caches design may keep dirty data and hence, error correction means are needed to keep failure rates low enough. However, tolerating faults in DL1 cache memories requires, in general, the use of Error Correction Codes (ECC) to allow recovering data that has been corrupted, which carries significant impact in either DL1 cache latency or design complexity. If DL1 cache data is delivered before correction, ECC does not impact the critical path and can be computed offline. However, upon the detection of an error, direct and indirect consumers of erroneous data are squashed and restore a correct state before resuming operation. In general, simple microcontrollers used for critical real-time systems lack such support. If cache data is delivered after correction, an additional stage is needed after loading data to compute ECC and validate whether data is correct. Thus, back-to-back execution of consumers after a load operation occurs with an additional delay (typically one cycle), which has non-negligible impact in performance.

In this Chapter, we present an alternative deployment of ECC in L1 caches for critical real-time microcontrollers aimed at mitigating the impact of ECC calculation in L1 caches. We propose a Look-Ahead Error Correction (LAEC) scheme that anticipates the whole DL1 access process by one cycle, thus allowing to eliminate any performance overhead whenever such anticipation is possible. In particular, for in-order cores common in critical real-time embedded systems (e.g. LEON4, ARM Cortex-R5 cores), whenever the input address registers are not computed by the immediate predecessor instruction of a load instruction and the predecessor instruction is not a load instruction, we can perform the address calculation, DL1 access, and ECC computation one cycle ahead of time. In this way, data can be delivered in the same cycle it would be delivered in a non-protected DL1 cache without anticipation. We note that the constraints that could preclude the effectiveness of our mechanism occur seldom, thus allowing LAEC to achieve a performance close to that of an error-free processor without ECC, while outperforming designs that require an additional cycle before delivering error-free data.

We have evaluated LAEC by implementing it with Single-Error Correction Double-Error Detection (SECDED) in the DL1 cache of a cycle accurate processor model of the LEON4 [59]. Our results show that our look-ahead error correction scheme outperforms the baseline read-and-correct scheme by 6% on average across EEMBC Automotive [127] benchmarks, and is within 3.9% the performance of an ideal error-free design without any ECC support.

7.2 Motivation

Current processor designs for critical systems employ different approaches to include ECC schemes in caches. This is partially motivated by the fact that actual latency overheads depend on the particular ECC technique employed. For instance, using a parity bit is the simplest and fastest technique, and SECDED is more complex and slower. In general, processors targeting safety critical systems require having the ability to recover from faults which forces processor designers to architect solutions able to achieve that goal. As shown in Table 6.2 (from previous Chapter 6) processors available in the market use different approaches to protect caches from errors. For instance, the LEON family of processors advocates for using write-through caches with parity in the L1. The Freescale PowerQUICC offers the user the possibility to configure L1 caches as write-through or write-back restricting recovery capabilities to write-trough configuration only. They pay the costs in contention to reduce faults, since in the space domain these are more common. Finally, in the Arm Cortex family the processor IP is sold with the possibility of implementing both write policies and allowing using ECC or parity in L1 caches. However, as acknowledged in the datasheet [15], using ECC in the L1 can impact the maximum operating frequency of the processor. In this case the final decision on whether to tradeoff performance for reliability is left to the integrator.

7.2.1 Correcting Errors in Write-Through (WT) DL1 caches

A practical solution typically found in processors for critical system consists of using a cache hierarchy with inclusive caches and write-through (WT) policy in the DL1 [59, 136, 16]. A commonality in these designs is to include a parity bit in DL1 caches and SECDED in the L2 cache since the relative impact of latency overhead of SECDED in the L2 is lower. The main reason is that, even if L2 read hit latencies are increased due to the introduction of SECDED, its impact in overall performance is low due a two-fold reason. First, L2 read accesses occur seldom, and second, having an additional L2 cycle causes limited impact due to the already high L2 access latencies to send requests, access the L2 itself and return data read to the core. Overall, this configuration (DL1 parity + L2 SECDED) ensures that errors can be detected with the parity bit with virtually no impact in latency in the DL1, and recovered with the SECDED mechanism implemented in the L2.

While configurations using WT caches offer a workaround to the problem of correcting data in the DL1, this configuration presents the drawbacks that are inherent to the use of WT caches such as lower performance and higher energy consumption since every store operation is always propagated from the DL1 to the upper levels of the memory hierarchy (i.e. hardware shared resources). To mitigate this issue processors may include a store-buffer and/or use an L2 cache implementing a write-back (WB) policy. However, it has been shown [23] that performance guarantees (WCET estimates) on multicore processors incorporating WT caches are quite poor when compared with their WB counterpart despite implementing store-buffers and a WB L2 cache. This result is especially important since processors targeting critical systems do not only require guaranteeing reliable operation, but also offering high performance and time-predictable behavior [77], which calls for multicore processors implementing ECC in WB DL1 caches in an efficient manner.

7.2.2 Correcting Errors in Write-Back (WB) L1 caches

WB policies do not update, on a DL1 hit, the upper levels of the memory hierarchy. Hence, in our setup modified data can reside exclusively in the DL1, so using a WB policy requires implementing error correction capabilities to recover from errors in the DL1. However, as explained before, this has generally an impact in the access time to DL1 cache. Several approaches exist to deal with the increase in DL1 access latency and they are implemented in commercial processors already. The particular processor architecture, the target operating frequency, and the manufacturing technology determine when to use one approach or the other. Below we describe four existing approaches:

1. **Decrease the operating processor frequency** is the most trivial approach to allow SECDED in the DL1 so that the error correction process can be accommodated within the last cycle of the cache access. However, this has a significant impact in the performance of the system. Some commercial processors for which the targeted operating frequency is sufficiently low or whose critical path is determined by other components may opt for this solution [15].

2. **Extra cache cycles.** Adding extra (non-pipelined) cycles in the DL1 access so that ECC computation fits in the L1 cache access time without impacting operating frequency. However, such a solution virtually doubles the time utilization of the DL1.

3. **Extra stage.** Pipelining cache accesses such that instructions proceed normally, adding a final cycle for ECC computation. Pipeline stalls will be introduced in the case of data dependencies (i.e. an instruction requires data for which ECC computation is not yet performed). The delay of the logic that detects and corrects errors can vary depending on the number of bits corrected. For SECDED, considered in this Chapter, this latency is smaller than an DL1 cache access [52, 143], and thus fits in a single additional cache cycle or stage pipeline.

4. **Speculate and flush.** Using a speculate and flush approach consists of processing accesses and delivering unchecked data, which may be used in parallel with ECC computation. Whenever the result of the ECC determines that the propagated data was erroneous, the pipeline is flushed or some instructions squashed, and a previous correct state needs to be recovered.

From these four approaches, we discard the former due to its noticeable performance degradation, and the latter due to the implementation complexity required to implement a flush mechanism in simple microcontrollers for critical systems like the ones we target. The extra stage and cache cycles solutions offer acceptable cost and implementation complexity trafeoffs, and will be the reference policies we compare our proposal with. Our proposal builds upon the Extra stage one, but anticipates the load access and ECC computation whenever possible so that no additional stalls are introduced due to ECC computation. In the next section we introduce details of the baseline approaches and present our look-ahead scheme.

7.3 Look-ahead Error-Correction

We propose an alternative approach to deploying ECC in DL1 caches. It consists in anticipating one cycle the address computation, the load access, and the ECC computation. This can be done when no data or structural dependence with older instructions occurs. Effectively anticipating one cycle the processing of DL1 load hits allows anticipating ECC computation by one cycle too. As a result, an instruction dependent on the loaded data can be executed back-to-back with the load without experiencing any delay due to ECC computation.

In this section, we first introduce the mechanism used to anticipate the address in our Look-Ahead Error-Correction (LAEC) proposal. Then, we describe the processor model on which the implementation and experiments will be conducted. Finally, we describe the implementation details of Extra Cache Cycle and Extra Stage approaches, as well as LAEC.

7.3.1 Address anticipation mechanism

There are several ways to predict the address of cache access. For instance, cache designs could incorporate a predictor similar to the ones employed in hardware data prefetchers [137]. However, since the focus of this Chapter is deploying ECC in relatively simple processors, we opt for an alternative method to predict the next DL1 access address. LAEC avoids mispredictions by anticipating address calculation only when it is guaranteed that such anticipation will deliver correct results.

In particular, LAEC avoids speculating on the address to prevent unnecessary accesses to the DL1. We anticipate address computation by reading the base register one cycle earlier if it has not to be modified by any previous instruction. This allows the address of the access to be computed one cycle earlier using an adder to add the register and the offset. LAEC also requires including two extra ports to the register file to retrieve the registers one cycle earlier, but in general an in-order single-issue processor has a small register file with few ports, so this would incur low power cost. In fact, it has been shown that energy is largely dominated by cache memories, so the energy consumption of the register file is small [106]. An access look-ahead can be performed when following two conditions hold:

1. *No resource hazard.* Since we anticipate the DL1 access and ECC computation by one cycle, we may conflict with the previous instruction if it accesses the DL1 simultaneously with the anticipated instruction. This occurs when the previous instruction is a non-predicted (i.e. branch speculated) load.
2. *No data hazard.* When the instruction prior to the load produces the address register of the load, we cannot anticipate the address computation. This is so because the input data for the ongoing instruction (load) is not yet ready whenever we want to anticipate its execution.

If none of these hazards occur, then we can compute the address, access DL1, and compute the ECC one cycle ahead of time. With *no resource hazard*, we guarantee that the DL1 read port is available. With *no data hazard* we guarantee that we are loading the right data, so no misprediction can occur and there is no need to flush.

7.3.2 Processor Model

In order to implement LAEC in the DL1, we use a system resembling the NGMP [59]: a multicore processor that includes 4 single-issue in-order pipelined cores with L1 private caches and a shared L2 cache. In the NGMP, error recovery is guaranteed by using WT DL1 caches with a parity bit and implementing SECDED in the WB shared L2. However, to implement LAEC we modify the baseline implementation to include a WB DL1 cache. Note that this modification is already in the roadmap of the LEON processor family whose providers have already announced LEON5 processor implementing WB DL1 caches [43]. Also, using a WB DL1 cache has already been shown effective for this setup [23].

The original NGMP system has a seven stage pipelined design (see Figure 7.1). The memory stage uses a write buffer (not shown in Figure 7.1 for simplicity) where

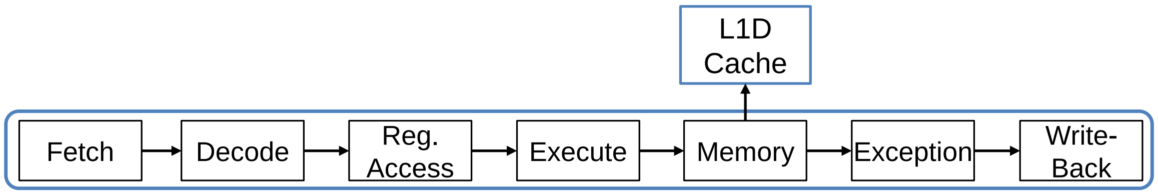
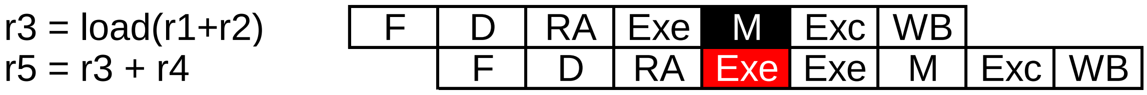


Figure 7.1: Baseline NGMP-like processor pipeline

all writes are stored until they can access DL1. A load that misses in DL1 blocks the pipeline. All loads stall the memory stage until the write buffer is empty to avoid consistency issues. Writes also stall the pipeline with backpressure when the write buffer is full, until it gets completely empty.



r3 = load(r1+r2)	F	D	RA	Exe	M	Exc	WB		
r5 = r3 + r4		F	D	RA	Exe	Exe	M	Exc	WB

Figure 7.2: Chronogram of a data dependency stall on the NGMP

In Figure 7.2 we show an example of two consecutive instructions with a data hazard between them that results in a 1 cycle stall for the younger instruction. In red we show the stage in which the young instruction stalls and in black the stage where the DL1 cache is accessed. In this case, it matches the memory stage, but this is not always the case in our proposed approach.

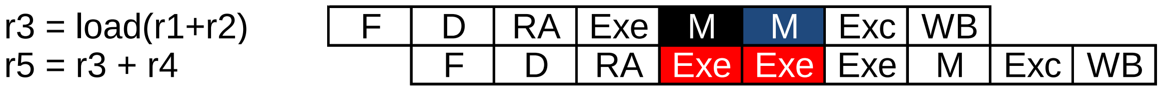
Next, we present the implementation details for existing Extra Cache Cycle and Extra Stage solutions as well as for LAEC. We also show how they could be implemented in a processor like the NGMP.

7.3.3 Extra Cache Cycle Implementation

A first simple approach that would require little changes to the current architecture is to make the ECC check in the memory stage so that this stage spans across two cycles, thus increasing the latency of a load hit from 1 cycle to 2 cycles.

In terms of hardware cost and implementation complexity, besides the ECC logic and its associated array in the DL1, little extra logic is needed in order to stall earlier processor stages (those before the memory stage), since stall logic already exists to stall the pipeline upon a DL1 cache miss.

The performance impact that this solution can have is relatively high, since it will double the cycles in the memory stage for DL1 hits.



r3 = load(r1+r2)	F	D	RA	Exe	M	M	Exc	WB		
r5 = r3 + r4		F	D	RA	Exe	Exe	Exe	M	Exc	WB

Figure 7.3: Data dependency stall with Extra Cache Cycle

Figure 7.3 extends the example in Figure 7.2 when the Extra Cache Cycle solution is applied. Now the young instruction that depends on the loaded data needs to stall one additional cycle for the value to be both loaded and checked. The stage in blue performs the ECC computation, which is done on the second cycle of the memory stage.

7.3.4 Extra Stage Implementation

Another simple approach would be to add a new pipeline stage after the Memory one: the ECC stage. This stage would compute the ECC for DL1 load hits, and compare it with the existing value stored in its ECC array. For writes that hit, it would compute the new ECC and store it in the ECC array.

In terms of timing, this solution can stall the pipeline when a load that hits in cache is followed (distance 1 or 2) by an instruction that uses the loaded value. In particular, the instruction immediately after the load cannot use the loaded value because its execute stage overlaps with the load memory stage. The second instruction starts its execution stage right after the load fetches the data from DL1 on a cache hit. Hence, if the second instruction after the load was allowed to use the loaded value as a source operand before computing its ECC and this value was incorrect, a complex recovery mechanism would be required to restore the processor state to a previous correct state. Instead, in our implementation, if this scenario happens, the processor stalls to avoid continuing the execution with a potential incorrect value.

It is worth noting that this only happens for loads that hit on the DL1 cache. For loads that miss, and have to request the data to higher levels (L2 or memory), the ECC of the data is checked in the corresponding cache level or main memory, so there is no need to check it again in the new ECC stage. For stores, the write buffer is usually enough to hide this latency.

In addition to the ECC logic and ECC array in the DL1, small extra hardware is needed in order to stall the stages before the Memory one. However, as explained before, this logic already exists to manage DL1 load misses.

In terms of potential impact, this solution is affected negatively by the number of times that an instruction consuming the loaded data is stalled due to the ECC stage, which can occur often since it is common having a consumer for loaded data in the range of the 2 following instructions.

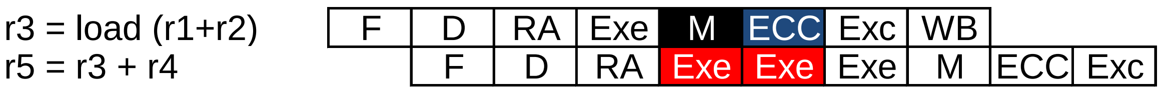


Figure 7.4: Data dependency stall with Extra Stage

Figure 7.4 shows a scenario similar to that of the Extra Cycle solution. The young instruction needs to stall for 2 cycles due to the data hazard. The advantage over the previous solution is shown in Figure 7.5: when there is no data hazard, consecutive instructions can continue execution without a stall, since Memory and ECC are pipelined.

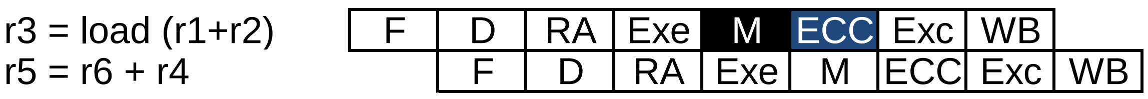


Figure 7.5: No data dependencies with Extra Stage

7.3.5 LAEC implementation

LAEC anticipates the load and ECC computation by 1 cycle. To that end, the address registers are read one cycle earlier, so two additional read ports are required in the register file. If any of the registers has been generated but not yet stored in the register file, it can be obtained from existing bypasses. Since the address computation for the load needs to be performed one cycle earlier (RA stage), an additional adder is also required (see Figure 7.6). We have checked with CACTI [115] the access times of a register file and an DL1 cache like the ones found in the LEON4 [59] (1088 bits for the register file, 16KB for the DL1 in 65nm). The difference between both is enough to include a 32 bit adder [8], so this addition does not increase the stage time of the memory stage.

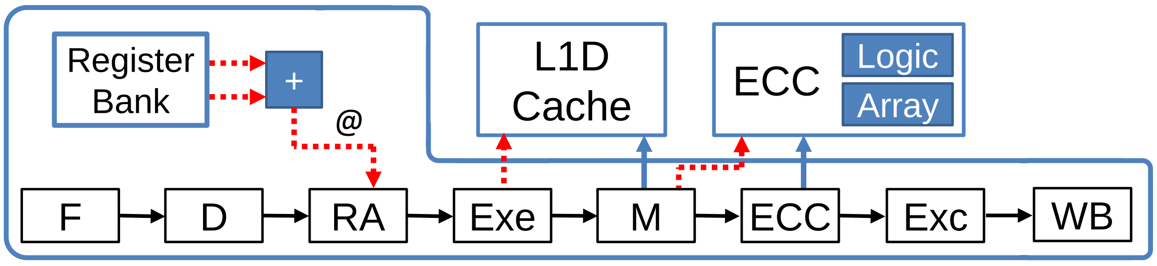
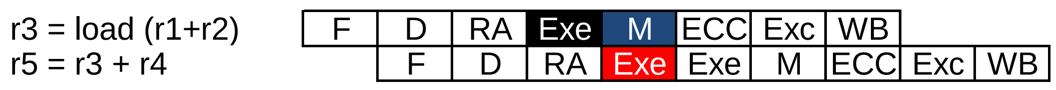


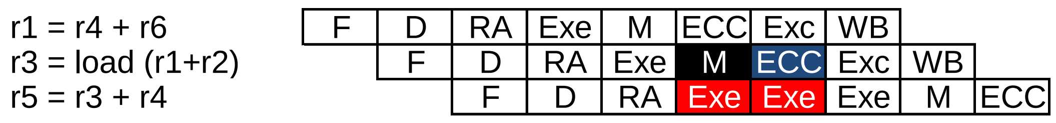
Figure 7.6: Modified NGMP-like processor to support ECC using LAEC

If the previous instruction generates one of the source operands of the load instruction, this technique cannot be used, since it would require the operands a cycle earlier than they are available. In this case, the processor operates normally (like in the Extra Stage implementation), with no look-ahead. Then, if any of the 2 instructions right after the load requires the loaded data, there will be a cycle penalty due to the address not being previously computed. Analogously, if the previous instruction is a non-predicted load, it will require the DL1 port (memory stage) simultaneously with the current anticipated load. In this case, the current load cannot be anticipated due to a resource hazard. These two scenarios are the only ones where our solution can introduce a penalty in execution time. This means that LAEC always performs equal or better than the Extra Stage implementation since, in the worse case, it cannot anticipate the load and just operates the same way as the Extra stage.

Figure 7.7 (a) shows a scenario where the added ECC penalty cycle can be avoided because of prediction. Registers r1 and r2 are both read and added on the RA (Register Access) stage. Then, on the Exe (Execution) stage, the DL1 cache is accessed. Afterwards, on the M (Memory) stage the ECC is computed. This results in the loaded data being ready to the younger instruction without additional penalty when compared to the baseline no-ECC solution (Figure 7.2). Conversely, Figure 7.7 (b)



(a) Look-ahead on LAEC



(b) Normal (no look-ahead) execution on LAEC

Figure 7.7: Possible look-ahead and normal scenarios with LAEC

Table 7.1: Cache redundancy: performance impact of existing approaches

	a2time	aifftr	aifirf	aiifft	basefp	bitmnp	cacheb	canrdr	idctrn	iirflt	matrix	pntrch	puwmod	rspeed	tblook	ttsprk	average
% of hit loads	89	97	90	97	84	98	77	86	92	86	99	90	85	84	88	84	89
% of dep. loads	68	53	66	54	80	65	13	67	59	63	64	61	66	66	68	61	60
% of loads	23	21	26	21	24	20	18	29	21	26	20	25	31	29	29	31	25

shows a scenario where there is still penalty due to a data dependency that prevents the look-ahead. Now the load is preceded by an instruction that computes one of its source registers. This means that r1 is not ready in the RA stage, so a normal execution (DL1 access on the M stage and ECC computation on ECC stage) is performed, resulting in a 1 cycle extra stall.

The schematic of the modifications required is shown in Figure 7.6. Note that the DL1 cache can now be accessed in two different stages: Exe or M. It will be accessed in Exe when there is a look-ahead (in red) and in M when there is not (in blue). Likewise, the ECC logic and array can also be accessed in two different stages: M (for look-ahead, in red) and ECC (normal execution, in blue). Since the baseline processor already includes most of the control logic needed for this solution, as well as bypasses from the desired stages, there is no significant cost in terms of hardware. The only explicit changes apart from the logic are a 32-bit adder and two extra read ports on the register file. Overall, implementing our LAEC proposal in an NGMP-like processor incurs in low hardware cost and implementation complexity.

7.4 Evaluation

We use the simulator setup described in Section 3.3 and the configuration of subsection 3.2.4. As benchmarks we use EEMBC Automotive [127], introduced in Section 3.7. The simulations are run in a multicore system but only a single core executes a task, since the focus of this work is on core performance.

The overhead of the existing approaches is due to stalls that happen when there

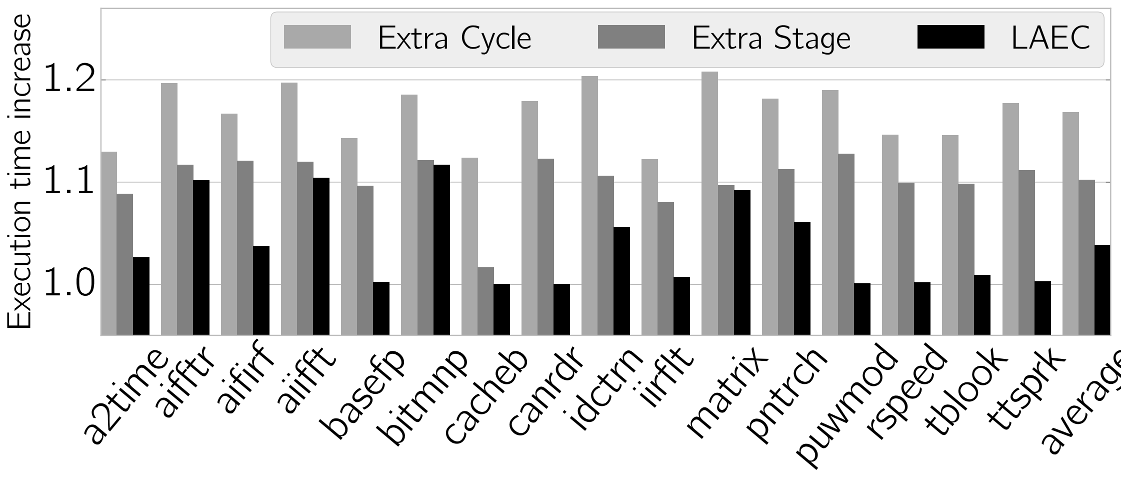


Figure 7.8: Execution time increase of the different solutions compared to the baseline no-ECC system.

is a DL1 hit that has a data dependency with the preceding instruction.

The first row in Table 7.1 shows the percentage of load instructions that hit in the DL1 cache. We see that with an average of 89% hit rate, most of the loads generate hits in cache, hence potentially generating stalls. The second row shows the percentage of load instructions followed, at a distance 1 or 2, by an instruction that uses as a source operand the loaded data. On average 60% of the loads cause a stall. Finally, loads represent between a 20% and 30% of all the executed instructions, significant enough to impact performance.

7.4.1 Experimental results

Figure 7.8 shows the increase in execution time with respect to a no-ECC system. Extra cycle shows the highest performance degradation, with a 17% execution time increase on average w.r.t. a configuration without ECC stage, reaching up to 20% for some benchmarks (`aifftr` and `matrix`).

Extra stage shows around 7% less performance degradation than Extra cycle, with a 10% on average. This occurs because its pipelined designed avoids some stalls. All benchmarks perform similarly, except `cacheb`. This benchmark shows little performance degradation compared with the baseline no-ECC (2%). This is due to the number of loads that are followed by dependent instructions. While in rest of benchmarks between 50% and 80% of the loads have this property, in `cacheb` just 13% of the loads have it. This results in fewer cases that stall the pipeline an additional cycle, and thus in lower performance degradation.

Finally, LAEC, due to its anticipated load execution, saves most of the stalls. On average, LAEC increases execution time less than 4%, being such increase below 1% in several benchmarks such as `basefp`, `cacheb`, `canrdr`, `puwmod`, `rspeed` and `ttsprk`. Out of the two potential conditions that LAEC needs to meet to cause a stall (resource and data hazards), most of them are due to data hazards. That is, the scenario where an instruction generates the address to be loaded, the next instruction performs the load that hits in cache, and the next 1 or 2 instructions consume the loaded data (as shown in Figure 7.7 a)). There are four benchmarks (`aifftr`, `aiifft`, `bitmnp`, `matrix`) that show almost no improvement when comparing LAEC with Extra Stage. This is because most of the loads that have dependent instructions executed right after, so causing stalls for Extra Stage, also have their source operand produced by the previous instruction, which prevents load anticipation and causes stalls for LAEC.

In terms of power, the proposed solution has minimal impact (less than 1%). However, since the execution time is increased, leakage energy consumption increases proportionally to the increase in execution time. This means that for extra cycle and extra stage, leakage energy consumption increases by around 17% and 10% on average; and for LAEC by less than 4%.

Note that, as explained before, while compiler optimizations could help mitigating stalls, they are normally forbidden in critical software due to traceability between source and binary files needed for certification. Moreover, those systems often execute legacy code where no binary modifications are possible. On average, LAEC shows a 13% decrease in performance degradation when compared to Extra cycle and a 6% decrease compared to Extra stage.

7.5 Related Work

The most common microarchitectural solution to error correction relies on the use of parity or error ECC [42] to detect and correct errors. Parity suffices for read-only caches (e.g. instruction caches) and write-through caches. When data can be dirty, then ECC is required.

Several works aims at providing support for both, permanent and transient faults. Those works often consider high permanent fault rates due to low voltage operation, and propose mechanisms to tolerate those faults while providing resilience against transient faults. Some works propose disabling faulty entries at different granularities [109, 3], potentially setting up spare cache lines to replace faulty ones [91], or combining faulty entries to form fault-free ones [160, 89], potentially combining these designs with heterogeneous ECC depending on the faultiness of cache lines [159].

Some authors combine fault tolerance in caches with real-time requirements by ensuring that ECC guarantees that non-correctable permanent fault rates are below specific thresholds [107, 108], or by guaranteeing that spares (in the form of a victim cache) suffice to guarantee sufficiently low non-correctable permanent fault rates [6].

Some techniques target specifically soft errors. Some authors propose early evicting dirty cache lines to mitigate the probability of uncorrectable errors due to multi-bit upsets (MBUs) in caches with single-error-correction capabilities [162]. In our work we do not consider MBUs since technologies used in critical real-time systems are intended to suffer sufficienly low MBU rates. In any case, this concern is orthogonal to our work. Coarser-grain solutions such as lockstep execution are also common in critical real-time systems [76, 75]. However, those designs often combine also lockstep execution with ECC protection in caches, as in the case of the LEON3FT processor for the space domain [60], thus being orthogonal to our approach.

7.6 Conclusions

Emerging real-time applications require increased performance in embedded systems, but also enough reliability levels for specific domains. Write-back L1 caches can help increase performance of such multi-core systems, but dirty data requires additional error correction. Unfortunately, implementing ECC, such as SECDED, increases the end-to-end latency to fetch and correct data. This can result in significant performance degradation due to data dependencies between loads that hit in the L1 cache and the following (consumer) instructions. We propose a novel approach to mitigate this issue, called Look-Ahead Error-Correction (LAEC) that anticipates data loading by one cycle whenever possible to avoid potential stalls. Our results show that our technique improves performance by 6%-13% w.r.t. existing solutions, and is only within 4% of the ideal case where no ECC is needed. Our proposal not only has low execution time overhead but also low design complexity and hardware cost since no costly instruction flush and state recovery is needed.

Chapter 8

Prefetching

8.1 Introduction

Prefetching is an effective means to mitigate the impact in execution time of cache misses [84, 18, 83]. Prefetching mitigates the impact of large data retrieval latencies in program's execution time by fetching data, before it is actually needed. While prefetching is effective to increase average performance in many high-performance processors [147, 44], high-integrity systems are often precluded from using prefetching whenever it is not explicitly controlled by the programmer. In particular,when prefetches occur as a result of explicit instructions in the code (software prefetching), they can be analyzed virtually with analogous complexity to that of non-prefetch memory accesses. However, when prefetching is produced automatically by hardware, then it becomes an overwhelming complex task to provide reliable and tight Worst-Case Execution Time (Worst-Case Execution Time (WCET)) estimates for programs due to the difficulties to reason on the worst-case timing behavior, and to relate analysis and operation conditions in the case of Measurement-Based Timing Analysis (MBTA).

In this Chapter, we work towards leveraging the benefits of hardware prefetchers in a safe manner in high-integrity systems. To that end, we present a formal framework for the design of time-predictable cache hierarchies with hardware prefetchers (or prefetchers for short), thus compatible with the requirements on high-integrity systems. In particular, we show how some access sequences can be proven to *dominate* others, meaning that they lead necessarily to higher execution times in timing anomaly free processors. Furthermore, we elaborate on the dominance relationships upon *composition* of specific cache-related components (e.g. buffers, caches, prefetchers), showing that complex cache hierarchies with hardware prefetchers can be made time-analyzable by construction, thus enabling the reliable use of hardware prefetchers in high-integrity systems, reducing WCET estimates. Our results show that our proposed time-predictable prefetcher, for the EEMBC Auto benchmark suite, achieves average WCET reductions ranging from 10% to 39%, 28% on average, with respect to a memory system without prefetcher.

The rest of this Chapter is organized as follows. Section 8.2 provides some background on caches and prefetchers. Section 8.3 introduces our formal framework for the design and analysis of prefetchers. Section 8.4 presents specific realizations of our framework to build time-analyzable systems. Those realizations are evaluated in Section 8.5 in the context of a NGMP-like processor. Related work is presented in Section 8.6. Finally, we summarize the main findings of this work in Section 8.7.

8.2 Background

Large cache memories and multi-level cache hierarchies allow mitigating *capacity* misses [100], whereas high associativity mitigates *conflict* misses. Instead, *cold* misses can only be partially mitigated by using larger cache lines so on a miss more data, as much as they fit in a cache line, are brought to cache. Yet, cache capacity and memory bandwidth limitations make that cache lines rarely exceed some tens of bytes. Thus, many cold misses cannot be avoided by modifying cache characteristics, which leads to poor performance for programs with low data reuse.

Prefetching reduces the delays caused by cache misses by fetching data in advance before it is actually needed [102]. Prefetching techniques trade off several factors including (1) fetching data early enough not to cause program stalls, (2) fetching data late enough to keep storage demands low, and (3) fetching low amounts of data to reduce memory bandwidth needs hence avoiding side effects on on-demand accesses.

High-integrity systems either build upon no prefetching at all, or upon software prefetching [139, 49], in which prefetching occurs as a result of explicit instructions added to program's code. Simplifying cache predictability analysis. However, software prefetch suffers several limitations: (1) it can only be applied on operated data (code is excluded); (2) information on whether accesses hit or miss is unknown in many cases, which makes more difficult the insertion of prefetch operations; (3) end users (or compilers) have only static information to predict dynamic cache hit/miss behavior; and (4) prefetch operations are inserted explicitly in the code, limiting the particular data and when it can be prefetched.

Hardware prefetchers, widely used in high-performance processors [147, 44], circumvent the limitations of software prefetching boosting average performance. However, (hardware) prefetchers challenge timing analysis:

- Prefetch operations do not appear explicitly in the code, challenging the ability of timing analyses to predict their beneficial and detrimental effects, i.e. whether some data has been prefetched and whether some data has been evicted by prefetch operations.
- The particular data prefetched and when they are prefetched depends on dynamic information related to the timing of events. In general, timing varies in non-obvious ways between the analysis and operation phases of the system which, in the case of MBTA, normally defeats any attempt to relate measurements collected during the analysis phase with the timing behavior during operation.

Therefore, arbitrary (hardware) prefetcher schemes are not allowed in the context of critical real-time embedded systems if those prefetchers cannot be proven to adhere to specific properties to enable reliable timing analysis.

8.3 Formal Framework

We build our framework for the design of time-predictable hardware prefetchers upon the concept of component (C), which receives one or several input time-stamped address sequences ($I, O, Z...$), has an internal state S, and produces one or several output sequences. In the context of prefetchers and cache memories, input and output sequences consist of a series of tuples where each tuple u has two elements, $u{=}(u.a, u.t)$: an address $u.a$, and a timestamp associated to such address $u.t$. Sequences are sorted by the timestamp of their tuples so that:

$$Z =< u_1, u_2, ...u_n >, \forall i, j : 1 \leq i < j \leq n : u_i.t \leq u_j.t$$

A component C is queried with a input sequence I and produces an output sequence O, see Figure 8.1, where $I =< u_1, u_2, ...u_n >$, $O =< v_1, v_2, ...v_m >$ and u_i and v_j are tuples. C has an initial state $S = \{a_1, a_2, ...a_p\}$ where a_k is a (memory) address. The addresses in the internal state S refer to, in the context of caches, prefetchers or buffers, those addresses whose data is stored in the element.

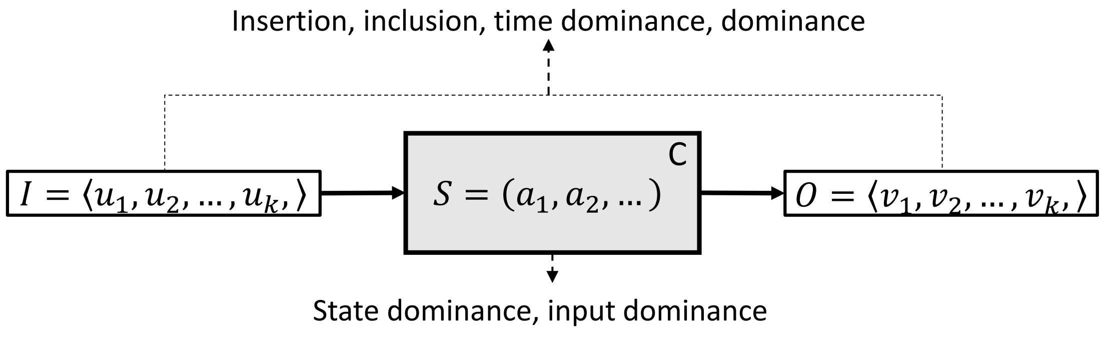

Insertion, inclusion, time dominance, dominance
$I = \langle u_1, u_2, \ldots, u_k, \rangle$
C
$S = (a_1, a_2, \ldots)$
$O = \langle v_1, v_2, \ldots, v_k, \rangle$
State dominance, input dominance

Figure 8.1: Formal framework sequences and componenents with their properties

8.3.1 Sequence Operators and Relationships

The **Insertion** operator when applied to a sequence Z and a tuple w produces a sequence Z' including all tuples in Z and w preserving the time order of tuples.

$$Z' = insert(Z, w) =< u_1, u_2, ..., u_i, w, u_{i+1}, ...u_n >$$

Hence, $u_i.t \leq w.t \leq u_{i+1}.t$. For multiple insertions, *insert* can be represented as follows for the sake of notation brevity:

$$Z'' = insert(Z, w_1, w_2, ...w_k) = insert(...insert(insert(Z, w_1), w_2)...w_k)$$

Inclusion captures whether a sequence Z_i is a subset of another sequence Z_j.

For instance, given $Z_1 =< u_1, u_2, ...u_n >$ and $Z_2 =< v_1, v_2, ...v_m >$, where $n \leq m$, $inclus(Z_1, Z_2)$ holds if all elements in Z_1 are included in Z_2.

$$inclus(Z_1, Z_2) \iff \forall i : 1 \leq i \leq n : u_i \in Z_2$$

Obviously, $inclus(Z_1, Z_1)$ and $inclus(Z_1, insert(Z_1, w))$ hold true. If several accesses to the same address X appear in Z_1 and $inclus(Z_1, Z_2)$ holds, then X appears in Z_2 at least as many times as in Z_1.

Time dominance. Given $Z_1 =< u_1, u_2, ...u_n >$ and $Z_2 =< v_1, v_2, ...v_n >$, with *exactly* the same addresses in the same order, Z_2 is said to time-dominate *tdom* Z_1 if each address in Z_2 has a timestamp equal or higher than its counterpart in Z_1.

$$tdom(Z_1, Z_2) \iff \forall i : 1 \leq i \leq n : v_i.a = u_i.a \ \wedge \ v_i.t \geq u_i.t$$

Dominance. Given $Z_1 =< u_1, u_2, ...u_n >$ and $Z_2 =< v_1, v_2, ...v_m >$, where $n \leq m$, Z_2 fully dominates Z_1 if there exists a sequence Z_1' that is time-dominated by Z_2 and Z_1' is obtained by performing zero or more insertions on Z_1. No constraints are put on the address or the time stamp of the tuples added.

$$dom(Z_1, Z_2) \iff \exists Z_1' : Z_1' = insert(Z_1, w_1, ...w_{m-n}) : tdom(Z_1', Z_2)$$

Whenever Z_2 has more than one access to the same address, the same principle applies with some particularities. For instance, let assume $Z1 =< (X1, 4), (X2, 10) >$ and $Z2 =< (X1, 5), (X2, 8), (X2, 12) >$. If $Z_1' = insert(Z_1, (X2, 8))$ then $dom(Z1, Z2)$ holds true as $X1 : 5 \geq 4$ and for all instances (access to) $X2$ it holds that $vi.t \geq ui.t$, in particular, $8 \geq 8$ and $12 \geq 10$. This would not be the case for $Z_1' = insert(Z_1, (X2, 12))$ since there is once X2 instance for which $vi.t \geq ui.t$ does not hold. It follows that dominance requires that Z_2 at least as many tuples as Z_1. That is, if $|Z_2| \leq |Z_1|$, Z_2 necessarily does not dominate Z_1.

8.3.2 Component Operators and Relationships

State dominance. Let us assume a component C with an initial state S_2. For a given an input sequence I, S_2 state dominates *sdom* another initial state S_1 if the output sequence under the former dominates the output sequence under the latter. That is

$$sdom(C, I, S_1, S_2) \iff O_1 = C(I, S_1) \wedge O_" = C(I, S_2) \wedge dom(O_1, O_2)$$

If a given initial state S_{max} dominates all other states for a given input I, then state dominance is generalized as follows, where $\mathbb{S}$ stands for the set of all potential initial states.

$$sdom(C, I, S*, S_{max}) \iff \forall S_i : S_i \in \mathbb{S} : sdom(C, I, S_i, S_{max})$$

State dominance can also be generalized for the case where a state dominates all others for any given input sequence, where $\mathbb{I}$ stands for the set of all potential input sequences.

$$sdom(C, I*, S*, S_{max}) \iff \forall I_j, S_i : I_j \in \mathbb{I} \ \wedge \ S_i \in \mathbb{S} : sdom(C, I_j, S_i, S_{max})$$

State dominance is of prominent importance for MBTA: by determining and enforcing S_{max} at analysis, WCET estimates hold during operation, when the initial necessarily lead to shorter execution times than those caused by S_{max}.

Input dominance is a component property such that given a sequence I_2 that dominates another sequence I_1, a component C with initial state S delivers a sequence O_2 for I_2 that dominates O_1 delivered for I_1. That is:

$$idom(C, I_1, I_2, S) \iff O_1 = C(I_1, S) \wedge O_2 = C(I_2, S) \wedge dom(I_1, I_2) \wedge dom(O_1, O_2)$$

Input dominance ensures that by using at analysis an input sequence I_{max} that dominates the input(s) sequence(s) of interest (e.g. I_j) together with S_{max}, so that $idom(C, I_j, I_{max}, S_{max})$, the WCET estimates at analysis hold during operation. This is so since any input sequence and initial state lead to shorter execution times than those caused by I_{max} and S_{max}.

It stands, therefore, that a key objective of the analysis is determining I_{max} and S_{max} so that WCET estimates obtained at analysis hold valid during operation.

8.3.3 Components with Multiple Input/Output Sequences

Definitions so far has focused on single-input single-output components. Those definitions can be extended to multiple-input multiple-output components, with some constraints:

Input dominance applies to *all* input sequences, and output dominance applies to *all* output sequences. For instance, in the case of state dominance, given $< O_1^1, ...O_n^1 >= C(< I_1, ...I_m >, S_1)$ and $< O_1^2, ...O_n^2 >= C(< I_1, ...I_m >, S_2)$, state dominance *sdom* for a given component C with inputs $< I_1, ...I_m >$ is defined as

$$sdom(C, < I_1, ...I_m >, S_1, S_2) \iff \forall i : 1 \leq i \leq n : dom(O_i^1, O_i^2)$$

Analogously, input dominance is also formulated as follows:

$$\begin{aligned} & idom(C, < I_1^1, ...I_m^1 >, < I_1^2, ...I_m^2 >, S) \iff \\ & \quad < O_1^1, ...O_n^1 >= C(< I_1^1, ...I_m^1 >, S) \ \wedge \ < O_1^2, ...O_n^2 >= C(< I_1^2, ...I_m^2 >, S) \ \wedge \\ & \quad \forall i : 1 \leq j \leq m : dom(I_j^1, I_j^2) \ \wedge \ \forall i : 1 \leq i \leq n : dom(O_i^1, O_i^2) \end{aligned}$$

8.3.4 Component Composition

Building upon the formulation above, we can compose components to conform a memory system. In particular, we focus on serial and parallel component composition, which allow building complex systems by applying recursively these basic composition methods.

Serial composition. Given components C_1 and C_2, we connect them serially so that $O_1 = C_1(I_1, S_1)$ and $O_2 = C_2(O_1, S_2)$, see Figure 8.2 (left). If we use an input sequence I_1' for C_1 that dominates I_1, then we will obtain an output sequence O_2' for C_2 that dominates O_2. That is, if $dom(I_1, I_1')$, then a component preserving input dominance $idom(C_1, I_1, I_1', S_1)$ guarantees that $dom(O_1, O_1')$. Therefore, if the second component also preserves input dominance $idom(C_1, O_1, O_1', S_2)$ it guarantees that $dom(O_2, O_2')$.

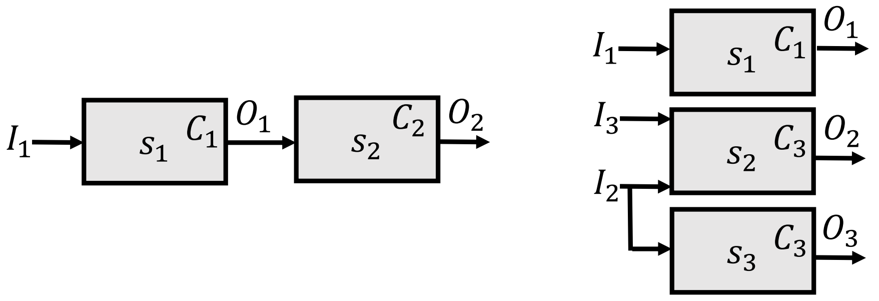


Figure 8.2: Formal framework sequential and parallel component composition

Analogous reasoning applies to multiple-input multiple-output components and state dominance, since sequence dominance properties become *transitive*. Thus, by composing serially components that preserve input dominance and state dominance, we can model them as a single component. The input sequences of that atomic component are those provided to all components excluding those produced by any of the components. The output sequences are those provided by all components excluding those consumed only by any of the components. For instance, we can define a serial composition of the following components:

$$\begin{aligned} < O_1^1, O_2^1 > &= C_1(I_1, S_1) \\ < O_1^2, O_2^2 > &= C_2(< I_2, O_1^1 >, S_2) \\ O_1^3 &= C_3(< O_1^2, O_2^2 >, S_3) \end{aligned}$$

where I_1 and I_2 for components C_1 and C_2 respectively are provided externally, and outputs O_2^1 and O_1^3 are given as external outputs. Externally, these component composition could be modelled as a single component as follows:

$$< O_2^1, O_1^3 >= C_{1+2+3}(< I_1, I_2 >, S_1 \cup S_2 \cup S_3)$$

Parallel composition. We can also components in parallel, see Figure 8.2 (right), that preserve input and state dominance. For instance, we could have the following parallel components:

$$\begin{aligned} O_1 &= C_1(I_1, S_1) \\ O_2 &= C_2(< I_2, I_3 >, S_2) \\ O_3 &= C_3(I_2, S_3) \end{aligned}$$

and building upon input and state dominance, we can prove that by using, for instance, an input I'_2 such that $dom(I_2, I'_2)$, then $dom(O_2, O'_2)$ and $dom(O_3, O'_3)$ hold. Note that O_1 remains unmodified, so given that domination preserves the identity function ($dom(O_1, O_1)$), the output of C_1 also preserves dominance. Thus, dominance holds for the output sequences of all components. As for serial composition, we can model them as a single component as follows:

$$< O_1, O_2, O_3 >= C_{1||2||3}(< I_1, I_2, I_3 >, < S_1, S_2, S_3 >)$$

Overall, since both serial and parallel composition of input and state dominant components allows modelling the resulting composition as a single input and state dominant component, they can be applied recursively for as many components as needed preserving their properties.

8.4 On Building Memory Systems

Building upon components that adhere to input and state dominance (ISD) properties, in this section we provide the semantics and approach to create a memory system that matches the needs of MBTA. In particular, the memory system must allow (1) reasoning about worst-case timing behavior, and (2) relating analysis and operation conditions.

To reach this goal, we first provide a way to describe typical types of hardware blocks in a memory system, e.g. caches and buffers, with our ISD components (ISDCs for short). Then we describe how to include hardware prefetchers on the memory system such that in can be modelled with our ISDC. We also cover implementation considerations for memory systems and how ISD properties can still be preserved.

8.4.1 Mapping hardware blocks to ISDC

Processor core. Cores interface memory by delivering a sequence of addresses to read/write from/to at specific times. The input sequence for the memory system, I, can be described as a sequence of tuples consisting of an address and a timestamp when the address is delivered to the memory systems. While tuples may contain other metadata, such as the operation type, we disregard such information for the sake of this discussion.

The answer provided by the memory system for read operations consists of the data requested. For write operations the answer may have either no answer or an acknowledgment of request reception. Either the case, we can attach an address to each answer identifying the memory location accessed as well as a timestamp when the answer is delivered back to the core. Thus, we can build an output sequence O for the memory sequence with the same form as those for our ISDCs.

We build the following assumptions:

1. Addresses in I are independent of the output sequence O, thus meaning that the core issues memory requests in order, and only timestamps in I may vary

due to different O sequences. For instance, this could be the case of a fully in-order core with no speculation which, therefore, issues those read and write requests dictated by load and store instructions in the strict order they are found and memory system behavior can only create varying stalls in the core progress which, in turn, can only alter the timestamps of the requests issued.

2. The core, and the other components considered in this work, is free of timing anomalies, so that an increased delay in the requests issued in I can only lead to an increased delay in responses in O, but never the opposite.

Overall, the processor core can be modeled as an ISDC with its output sequence O_{core} corresponding to the requests issued, its input sequence I_{core} to the responses from the memory system, and an empty initial state. From a memory address stand point, cores (excluding caches, scratchpads and the like that belong to the memory system) keep no state since they may store values temporary in the register file or internal buffers, but they are managed explicitly and without using their addresses. Hence, the core is modeled as follows:

$$O_{core} = C_{core}(I_{core}, \varnothing)$$

The core, which is the *initiator* ISDC, can produce multiple output sequences and receive multiple input sequences. The most common case would be that of separated data and instruction requests and responses.

Main memory is a *complete* container so that it contains the data of all addresses. Also its response time is constant and requests are processed in order. Its input sequence may come from the core or from another type of ISDC. Hence, we can model main memory as follows:

$$O_{mem} = C_{mem}(I_{mem}, \mathbb{U})$$

where $\mathbb{U}$ stands for the universe of all potential addresses within the address range of the memory. For instance, in a 32-bit machine with a single main memory, $\mathbb{U}$ includes the 2^{32} potential addresses of the system.

Main memory may have multiple input request sequences and multiple output request sequences. For instance, different components could generate input request sequences for different address memory ranges, and separate output response sequences could be produced to answer each of those components. However, in this case main memory could be better represented as separated main memory components, one for each address range, so that those memory components have a single input and a single output sequence.

Cache memories. Unlike main memory, caches have limited capacity and varying initial state so they store a subset of the actual address space. Caches are subject to implementation constraints that affect their state upon the processing of input sequences. However, implementation constraints are considered later while in this section caches are considered as unlimited containers.

Caches have at least one input sequence provided by a core or another ISDC sitting in between the cache and the core serially connected. Typically, caches have also at least another input sequence coming from memory or another ISDC sitting in between this cache and main memory serially connected.

Caches have typically two output sequences: one to respond requests coming directly or indirectly from the core and another to send requests to main memory or other cache memories in case of a miss. While this does not need to be this way necessarily, the usual cache model is as follows:

$$< O_{Li}^{1}, ...O_{Li}^{n} >= C_{Li}(< I_{Li}^{1}, ...I_{Li}^{m} >, S)$$

where cache C_{Li} has m input sequences, n output sequences, and an initial state S. We use the identifier Li since typically we instantiate caches as L1, L2, etc. depending on their location in the memory hierarchy.

Interestingly, unlike the core and main memory, caches may have varying initial states. Given an unlimited container and a population of addresses $\mathbb{U}$, the number of potential initial states include any address set S such that $S \subset \mathbb{U}$. We assume that, whenever a cache receives a request of an address not included in the cache state, i) it cannot serve it, so it either generates a request to another component, ii) responds with a negative answer (typically encoded in the metadata); or iii) simply ignores the request. While the former corresponds to the typical case, our model admits all three scenarios. In any case, we assume that a well-designed cache provides its fastest answer in case of a cache hit, and a cache miss cannot take shorter to be answered. Under such constraints, for caches of unlimited space, we can claim that $S_{max} = \varnothing$ (i.e. the empty state), since it is the state leading to the lowest number of cache hits regardless of the input sequences.

Queues and buffers are another form of container intended to temporarily hold requests and forward them, either "as they are" or transformed. For instance, a queue or buffer may simply hold requests and forward them after some delay at a given rate so that requests received as input are sent in order of arrival as output with an increased timestamp. As for caches, we assume that queues and buffers are unlimited in size and leave for later consideration implementation constraints.

Queues and buffers may have multiple input sequences (e.g. from several caches) and multiple output sequences (e.g. to different memories mapping different address ranges). The most common case corresponds to single-input single-output queues and buffers, which would be described as follows:

$$O_{queue} = C_{queue}(I_{queue}, S)$$

where S stands for the initial state including the requests not yet processed. For instance, a queue taking 3 time units to process each request may hold an arbitrary number of unprocessed requests, as well as up to one request that has been processed during 1 or 2 cycles already.

Determining S_{max} for queues and buffers with unlimited size would require assuming that they hold an infinite number of requests, which would be useless in practice since it would lead to an infinite WCET estimate. Therefore, in order to obtain reliable and meaningful WCET estimates, the initial state of queues and buffers needs to be carefully controlled at analysis and during operation so that operation conditions do not lead to higher execution times than those obtained with the initial state at analysis.

Other hardware blocks. While other hardware blocks could be described with our framework (e.g. scratchpads, interconnection networks), we do not further elaborate on additional components since, for the discussion in this work, the components already described suffice.

8.4.2 Memory Systems

A cache-less memory system. A simple memory system comprises a single core C_{core} and a main memory C_{mem}. Such a system is modelled in our framework by making $O_{core} = I_{mem}$ and $I_{core} = O_{mem}$, as depicted in Figure 8.3, where the initial state is shown within each component.

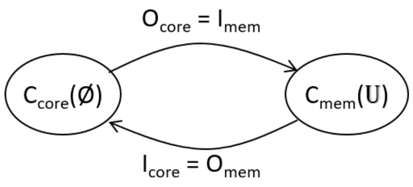


Figure 8.3: Simple system consisting of a core and main memory

A single-cache memory system comprises the core, a single cache memory (L1), and the main memory that are connected serially, see Figure 8.4. In particular, C_{L1} has two input and two output sequences connecting it to the core and main memory, and is described as follows:

$$< I_{core}, I_{mem} >= C_{L1}(< O_{core}, O_{mem} >, S)$$

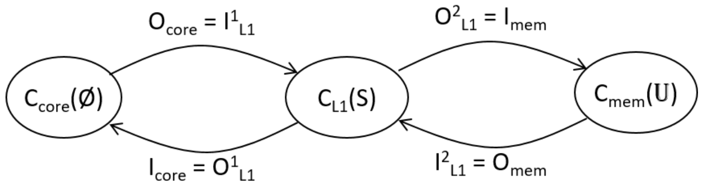


Figure 8.4: System consisting of a core, a single cache and main memory composed serially

As discussed before, we consider cache memories with unlimited size with shortest response for hits. Therefore, the initial state that dominates all other states, S_{max}, is the empty state. Hence, by enforcing S_{max} at analysis, we can estimate the WCET reliably for any set of input sequences.

A simple memory system with queues. In this case, core requests are sent to a queue C_{q1} to be later forwarded to main memory. Analogously, memory responses are sent to another queue C_{q2} to be later forwarded to the core, see Figure 8.5.

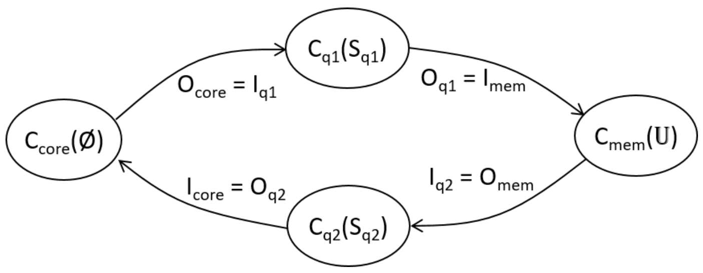
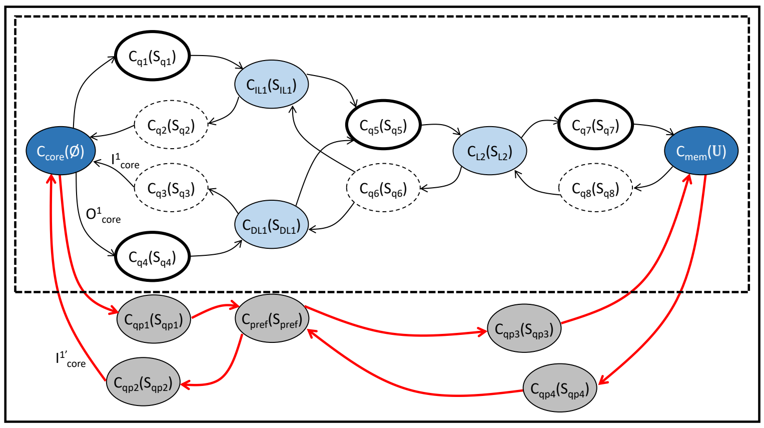
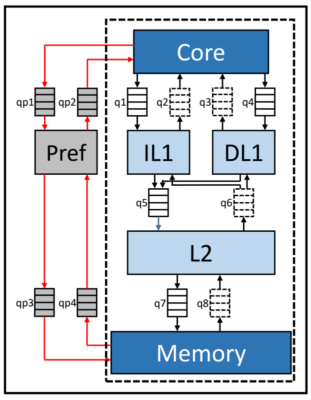


Figure 8.5: Single core, main memory and unidirectional queues composed serially



(a) Component-based representation.



(b) Architectural representation.

Figure 8.6: Complex system including a prefetcher component and its associated queues

In this case, determining the initial state S_{q1} and S_{q2} for the queues requires exercising some explicit control during operation to ensure that operation conditions are not worse than those at analysis. For instance, one could enforce empty initial states for all queues during operation, which would guarantee that any other state can only lead to higher response times for queues since they are ISDCs.

More complex system. Our framework allows describing more complex systems. For the sake of illustration, the part inside a dotted box of Figure 8.6 (both the component-based representation (a) and the architectural representation (b)) depicts a system consisting of a core (dark blue) that interfaces two first level caches (light blue), one for data (DL1) and one for instructions (IL1) with independent unidirectional queues. Queues interface a second level cache, L2, with shared unidirectional queues, so that incoming requests are received from IL1 and DL1 in one of the queues (C_{q5}) and responses are split across caches by another queue (C_{q6}). The L2 cache interfaces main memory (dark shaded) with independent unidirectional queues. Note that all queues managing requests in the way from the core to main memory are drawn with thick lines, whereas queues in the way from main memory to the core are drawn with dashed lines.

8.4.3 Including Prefetchers

Basic prefetcher. A set of sufficient conditions to include prefetcher in a memory system without affecting its key properties can be summarized as follows.

1. The prefetcher does not remove any content from caches.
2. The prefetcher uses separated queues.
3. The prefetcher is not placed serially with components of the original system.
4. Whenever it is necessary merging traffic between the original system and the prefetcher (e.g. in main memory), this must occur preserving dominance of the output sequence in the original system w.r.t. the output system in the system with prefetcher.
5. S_{max} for the prefetcher is the empty state, as for caches.

In order to illustrate how to meet the conditions above, we consider the base system in Figure 8.6 (the one inside the dotted box) to which we add a prefetcher component meeting the conditions above. The first step consists in describing the prefetcher itself, which we regard as a cache-like component whose fetch policy differs from that of regular caches. In particular, we define prefetchers as unlimited containers that, unlike caches, may issue requests upon hits and, also unlike caches, may request memory addresses different to those of the address in the input sequence (e.g. with a specific stride aiming to bring data needed in the future). Overall, we can define a prefetcher component as follows:

$$< O^1_{pref}, ...O^n_{pref} >= C_{pref}(< I^1_{pref}, ...I^m_{pref} >, S)$$

and we can include it in the base system as shown in the big box in Figure 8.6. In particular, we include C_{pref} as the prefetcher component, and 4 independent unidirectional queues (C_{qp1-4}), shown in grey in the figure. Their connections are shown with thick red arrows. As shown, one of the queues, C_{qp1}, receives a duplicated output of the core, O^1_{core}. At some point the prefetcher may produce requests that reach memory, C_{mem}, through queue C_{qp3}. Memory must be able to respond requests from the prefetcher (C_{qp3}) and from the original system (C_{q7}) in parallel without causing any disturbance in the output sequence delivered to C_{q7}. Finally, the core receives two sequences from one of the caches (DL1) and the prefetcher through the corresponding queues. The sequence from the cache, I^1_{core}, remains unaltered w.r.t. the original system as long as the core delivers the same output sequences to the components in the original system. However, the core also receives the sequence from the prefetcher, $I^{1'}_{core}$. In general, such sequence will include some of the addresses requested by the core with a lower timestamp than those delivered by the cache. Hence, the core will be able to use the earliest response, thus effectively merging the input sequence by obtaining the same addresses in the same order but with some potentially lower timestamps. Hence, by construction, the merged sequence is time-dominated by I^1_{core}, guaranteeing that the timing behavior with the prefetcher is necessarily better than without it.

A second effect of the prefetcher relates to the existing dependence between the output sequences of the core and its input sequences, as explained in Section 8.4.1 for the core component. In particular, by receiving a (merged) input sequence time-dominated by the original sequence (e.g. with some data prefetched), the ISD core component necessarily produces output sequences that are also time-dominated by the original ones, in line with the transitivity property of ISDC explained before for serial composition. In other words, by receiving some data earlier from the prefetcher, the core can make progress faster and issue its requests to the cache system earlier. Thus, such time-dominated output sequences can further lead the other components sitting serially after the core to deliver sequences also time-dominated by those in the original system. This is a cascade effect that may make the whole system process requests earlier, hence sending requests to other components earlier, which in turn also process them earlier and so on and so forth. Hence, the circular serial composition of components leads to time-dominated sequences as long as any of its components leads to time-dominated sequences as a consequence of having a suitable prefetcher in place. As a result, WCET estimates obtained with the prefetcher are necessarily better than without it.

WCET estimation can be performed by enforcing a worst-case initial state for the prefetcher at analysis. However, prefetchers presented so far are modelled similar to cache components, though they do not produce requests under the same circumstances. In particular, to be able to guarantee that the same S_{max} exists for caches and prefetchers (the empty state), we must guarantee that the behavior of prefetchers can never be worse on a hit than on a miss. A simple way to reach this goal consists in issuing the same requests upon a cache hit as upon a miss. This guarantees that misses do not lead to a different (potentially better) state than hits. Note that, if we issue requests upon a miss (e.g. accessing address $X1$) that bring any address different to that in the input request (e.g. address $X2$), this could make such miss to $X1$ lead to future hits (to $X2$) that may be misses under $X1$ hit. This is not an issue for cache components since, upon a miss, they only request contents that would already be present in cache upon a hit, thus never leading to a different (better) state for misses than for hits.

Relaxed conditions. While the conditions described so far are sufficient for enabling the use of prefetchers in Critical Real-Time Embedded Systems (CRTES), they may be too demanding since they impose that the system with prefetcher leads to lower execution times than that without the prefetcher. In practice, this constraint is not needed as long as we *always* use the prefetcher, both at analysis and during operation. In that case, we only need that the initial state of the prefetcher at analysis leads to dominant sequences w.r.t. those with any other state.

The simplest strategy to reach this goal builds again upon the use of the empty initial state for caches and prefetchers since it is their S_{max}. In this case, we can simplify the integration of prefetchers and use them as shown in Figure 8.7, where we have exactly the original system but with the prefetcher integrated with the cache component. Given a cache that, upon an access (e.g. to address $X1$) generates a sequence of requests O_{Li}^{X1} (typically an empty sequence on a hit or a sequence with one request for $X1$ address on a miss), and a prefetcher that upon the same access generates a sequence of requests O_{pref}^{X1}, the merged cache and prefetcher component issues a set of requests $O_{Li\cup pref}^{X1}$ as follows:

$$O_{Li\cup pref}^{X1} = O_{Li}^{X1} \cup O_{pref}^{X1}$$

Since sequences, as defined in this work, consist of tuples with addresses and timestamps, the union operator is applied to addresses. For instrance, let assume tha a given address appears in the cache and prefetcher sequences with different timestamps, which could be the case since each component produces it at a different time. The union of such two addresses would result in a single element with the common address and the timestamp of the slowest component.

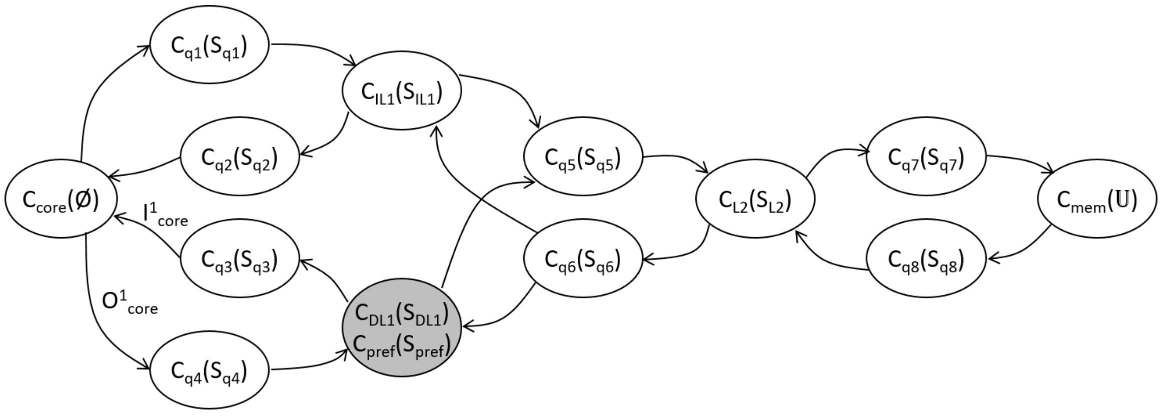


Figure 8.7: Complex system with prefetcher (relaxed conditions)

Hence, the merged output sequence will include addresses in O_{pref}^{X1} upon a hit, and *at least* addresses in O_{pref}^{X1} upon a miss. Hence, the sequence upon a miss can only be worse than that upon a hit.

$$dom(O_{pref}^{X1}, O_{Li\cup pref}^{X1})$$

By enforcing S_{max} (the empty state) as the initial state for the merged component, we enforce a state-dominant behavior for any input sequence. Any component C receiving as input the output sequence of the merged component receives, hence, an input-dominant sequence compared to that with any other initial state for the merged component.

$$idom(C, O_{pref}^{X1}, O_{Li\cup pref}^{X1}, S)$$

where S stands for the initial state of such other component.

Overall, under relaxed conditions the prefetcher does not remove any content from caches and S_{max} for the prefetcher is the empty state, as for caches.

8.4.4 Implementation Constraints

Limited-size Buffers and queues. Typically, in the case of buffers and queues, at least as defined in this work, entries are released as early as possible, so if the queue or buffer is full, there is no way of releasing entries faster and the new request cannot be admitted. Thus, limited-size buffers and queues produce backpressure when they are full. This, in our formulation, leads to increased timestamps that may back-propagate to an arbitrary number of requests.

Building upon time-dominance principles, we can argue that the increased timestamps for the input sequence of the buffer or queue imply that the input sequence for a limited-size buffer or queue is time-dominant w.r.t. that with an unlimited size. The arbitrary propagation of such backpressure, due to the transitive nature of the dominance properties in our framework, leads to time-dominant input and output sequences for all components connected directly or indirectly to those buffers and queues. Thus, imposing limited size buffers and queues does not break any of the properties of our framework.

Limited-size cache components impose the implementation of a form of replacement policy and, while not strictly needed, a form of placement policy, that is not independent of the replacement policy. For the sake of this discussion, we first stick to a limited-size set, which could be implemented as a fully-associative cache where we can potentially replace any element upon an eviction.

Our framework builds, explicitly or implicitly, upon several properties of unlimited-size caches that are impacted by using limited sizes. Those properties are as follows:

1. Fetching new data does not harm future requests since they create/generate no eviction.
2. Cache hits do not alter cache state.
3. The actual address accessed is irrelevant and it only matters whether the access hits or misses.

The first property (no eviction) cannot be preserved due to the finite capacity of the cache, which has some implications upon sequence dominance. Time-dominance remains unaffected since it imposes that request ordering is not altered and, for common replacement policies, the only relevant feature is the order of the addresses accessed. For instance, this is the case for Least Recently Used (LRU), first-in first-out (FIFO), Random Replacement (RR), and pseudo-LRU among others. Dominance, instead, may not hold for some replacement policies [26], such as LRU dominance. Let us illustrate this fact with an example. Given the sequence $S_1 =< (X1, 1), (X2, 2), (X3, 4), (X1, 5) >$ and a LRU cache with 2 entries, all accesses miss in cache since the access to address $X3$ evicts $X1$ and hence, the last access of the sequence is a miss. Now we create a new sequence $S_2 = insert(S_1, (X1, 3))$, thus obtaining $S_2 =< (X1, 1), (X2, 2), (X1, 3), (X3, 4), (X1, 5) >$. Since S_2 is built by inserting a tuple in S_1, S_2 should dominate S_1. However, with LRU replacement the second and third accesses to $X1$ become hits, so S_1 has 4 misses whereas S_2 has 3 misses and 2 hits. Thus, it cannot be proven that S_2 dominates S_1. If, instead, we use RR, it has already been proven that, by inserting accesses in a sequence, either the number of hits or misses increases, but none of them decreases [94]. This occurs because RR does not keep any replacement state information. Hence, dominance holds for RR caches.

Let us review this statement in more detail. When performing a new access to address X, there can be two outcomes depending on whether it hits or misses:

Hit: On a hit, the addresses in the cache remain unaltered, so future memory accesses will have the same outcome they would have had if w, where $w.a = X$, had not been inserted. Thus, the timing will be the same sequence of hits and misses as before inserting w, but with an additional hit. Hence, we have a sequence $S_1 =< u_1, u_2, ..., u_i, u_{i+1}, ..., u_n >$ and insert w such that $u_i.t \leq w.t \leq u_{i+1}.t$, thus meaning that w access occurs in between u_i and u_{i+1}. The resulting sequence, S_2 is obtained as $S_2 = insert(S_1, w)$. Therefore, $S_2 =< u_1, u_2, ..., u_i, w, u_{i+1}, ..., u_n >$, and hence, $dom(S_1, S_2)$, meaning that the new sequence S_2 dominates the original one S_1. Note that the insertion of an access in between u_i and u_{i+1} could potentially delay accesses after w by the latency of a hit access.

Miss: On an access miss to address X, we could have several possible outcomes depending on future memory accesses. Those cases depend on whether X is accessed later and whether the address evicted would have accessed later.

- Evicted address never accessed again.
  - The simplest case happens when the access inserted w, with $w.a = X$, corresponds to an address, X, which is never accessed again. In this case, we just insert a miss access, so the same reasoning as for the hit holds since the insertion has no impact on the remaining of the sequence, and accesses after w may get delayed by the latency of a miss.
  - Alternatively, we may have $S_1 =< u_1, u_2, ..., u_i, u_{i+1}, ..., u_{i+j}, u_{i+j+1}, ..., u_n >$ where $w.a = u_{i+j}.a = X$. In this case, we insert w, which incurs an extra miss, but a former miss (u_{i+j}) could become a hit. Hence, we insert w such that $u_i.t \leq w.t \leq u_{i+1}.t$, and where the following holds

$$\forall k : i + 1 \leq k \leq i + j : u_k.a \neq w.a$$

    meaning that no access in the subsequence $< u_{i+1}, ..., u_{i+j} >$ accesses address $w.a = X$. In this case, accesses in the subsequence $< u_{i+1}, ..., u_{i+j} >$ remain being hits and misses as before the insertion of w, so they may get delayed due to the added miss. However, access u_{i+j} becomes a hit, so accesses in the subsequence $< u_{i+j+1}, ..., u_n >$ may benefit from u_{i+j} being processed faster. In practice, those accesses experience a miss (w), the hits and misses in $< u_{i+1}, ..., u_{i+j} >$, and a hit ($u_{i+j}$), whereas in the original sequence they only experienced the hits and misses in $< u_{i+1}, ..., u_{i+j} >$ and a miss (u_{i+j}). Thus, the insertion of w would lead to an additional hit and the same number of misses in this case.

- Evicted address is accessed later. In this case, the eviction would cause an additional miss. Hence, the same casuistics as before hold and, additionally, an extra miss would occur. When serving such miss, it could potentially evict an address never accessed again, or an address that would be accessed again. In the latter case, a further additional miss would be caused and the reasoning would repeat until no additional misses are caused, which eventually occurs at the end of the sequence. Overall, Evicting an address that would be hit otherwise, can only lead to additional misses and no additional hit.

The second property (hits do not alter cache state), as shown in the example before, does not hold for some replacement policies such as LRU, since the LRU stack and hence, future replacement choices, are affected by hits. Instead, in the case of RR cache hits have no impact in cache state, thus preserving this property.

The third property (addresses are irrelevant) holds for fully-associative caches. However, set-associative caches and direct-mapped caches are sensitive to the actual addresses accessed since they determine the actual set accessed and hence, what address can be replaced upon a miss. We note, however, that, if dominance holds for all cache sets individually, then it holds for the cache as a whole. In this type of caches we would need to perform the analysis in a set by set basis, thus by splitting the input sequence across sets keeping for each set the tuples targeting addresses in that set. Then, by inserting an address in a sequence, this means that, in practice, it is inserted only in the sequence of the corresponding set, whereas the sequences for the other sets remain unaffected. Thus, if insertion preserves dominance in the affected set, dominance is preserved in all sets. As discussed before, this holds only for some replacement policies such as, for instance, RR, but not for some others such as LRU.

Let us assume the sequence $S =< (X1, 1), (X2, 2), (X3, 3), (X1, 4), (X4, 5) >$. Further assume that each address belongs to a set with modulo the number of sets. If we have 2 sets, then sequence S would be separated in two sequences, one for each set: $S_0 =< (X1, 1), (X3, 3), (X1, 4) >$ and $S_1 =< (X2, 2), (X4, 5) >$.

Then, each subsequence S_0 and S_1 would only affect sets 0 and 1 respectively. When inserting a new access in the global sequence S, it will only affect the set that it maps to. Thus, the sequence of the other sets will remain unaffected.

In summary, limited cache space does not affect the required dominance properties for RR caches, regardless of their associativity, but it does for other replacement policies such as LRU.

Another relevant aspect for caches is whether contents evicted are dirty (written) or not. In the case of write-through caches, all contents are clean and thus, no dirty contents can exist. However, write-back caches, where write operations are not forwarded to the upper memory level, are common. In general, on a miss, the cost of evicting a dirty line is higher than that of evicting a clean line since a clean line can be simply invalidated, whereas the dirty eviction requires a write operation to the upper memory level. Therefore, for a write-back cache, the empty cache is not its S_{max}. Instead, S_{max} corresponds to the case of a cache full of dirty useless contents unless some control is exercised both at analysis and operation to guarantee that analysis conditions can only lead to worse execution times than those during operation. Since, as discussed before, RR caches guarantee the properties needed by dominance relationships, we need to devise a valid strategy for these caches. In particular, we see several alternatives:

- Use only write-through caches and $S_{max} = \varnothing$.
- Build upon some hardware support to enforce all cache lines be marked as dirty for addresses neither used by the program under analysis nor with any functional impact in the system since they will be written back upon cache evictions.
- Make the system enforce all dirty cache lines be written back upon program execution completion so that any program finds the cache with no dirty contents. In this case, $S_{max} = \varnothing$ is the dominant state.

Prefetcher components can be managed identically to caches, although allowing them to hold dirty contents brings unnecessary complexity. Instead, the wisest solution is using prefetchers as fast memories for caches so that, upon a cache miss and prefetch hit, contents in the prefetcher are transferred to the cache and invalidated from the prefetcher. By operating this way, $S_{max} = \varnothing$ is the dominant state for the prefetcher.

Some prefetcher implementations use, instead of a separate storage space, the cache itself to store data prefetched. While this may be efficient in terms of storage, it may pollute cache contents, thus challenging dominance properties and so, the assumptions upon which our framework builds. Thus, a sufficient condition to avoid such an issue is keeping separated storage space for the prefetcher.

Prefetchers balance bringing as much useful contents as possible and making an efficient use of the memory bandwidth. As explained before, prefetch choices must not discriminate among hits and misses since, otherwise, this may create timing anomalies where a cache miss prefetches useful data that ends up leading to some hits and thus, a shorter execution time than having a hit in the original access. Instead, filters to limit the number of prefetches issued must be based on other parameters independent of the hit/miss behavior of accesses such as, for instance, prefetching only for some addresses (e.g. those addresses that follow specific patterns), prefetching periodically (e.g. once every 4 accesses), and the like.

For illustration purposes, we will use a prefetcher with separate storage that, upon an access to address A, it fetches the following cache line in memory on every cache access. This policy guarantees that each cache access produces exactly one prefetch request. While this mechanism can be improved to obtain more prefetching hits, it complies with the necessary restriction of producing the same access sequence regardless if the accesses hit or miss in cache. We leave for future work prefetcher optimization (in terms of hits and memory bandwidth) for prefetchers adhering to the framework constraints.

Time compositionality. Our proposal, similar to time compositionality [65], relies on separating the system in simpler and analyzable components. However, the key difference between both proposals is that time compositionality is used to simplify the timing analysis of a system, whereas our proposal takes into account the relationships between two states of a component, and use those relationships to

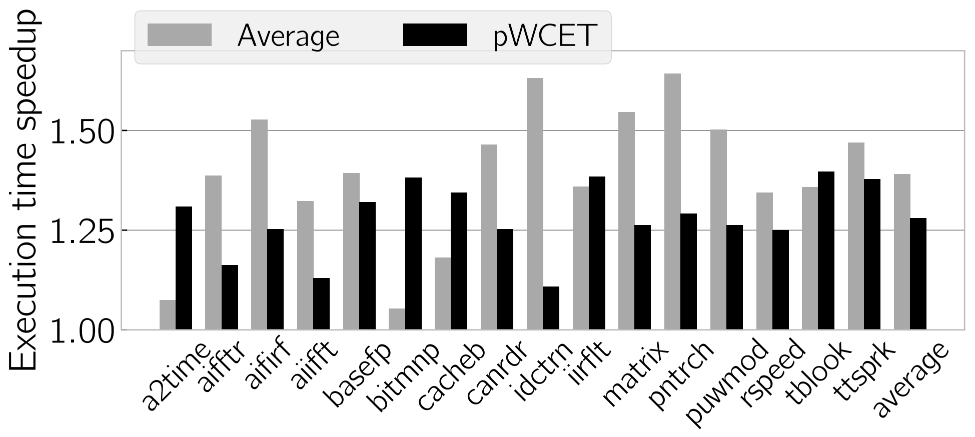


Figure 8.8: Prefetching results: average execution time and Probabilistic Worst-Case Execution Time (pWCET) speedup

enable the use of complex components (such as prefetchers). In our framework we extend the composability analysis to consider also the case of two components that interact between them in terms of functionality and timing. Our framework builds around this case to specify the constraints that the relationship between caches and prefetchers must preserve.

8.5 Evaluation

Architectural Simulator: We use a cycle accurate simulator that models a basic core like the one described in subsection 3.2.4. To model the system used by the framework described in this work, we make all instructions that do not miss in cache take 1 cycle to execute. Since the prefetching mechanism and our framework are independent of core operation, more complex cores (free of timing anomalies) could be used and the framework and methodology proposed would still be valid.

Benchmarks: We evaluate all benchmarks of the EEMBC automotive [127] suite as described in Section 3.7.

8.5.1 Realistic prefetcher system

For space constraints we omit the results of the base prefetcher and focus on the results of the more interesting realistic prefetcher, whose setup has limited size caches and prefetchers. In the case of the caches, we use 16KB 4-way 32B/line first level data cache (DL1) and instruction (IL1) caches, and a 256KB 4-way 32B/line second level (L2) cache. We also use a 4 entry (cache line) prefetcher that fetches the following address (+1). Given the expected short reuse time, with just 4 entries we achieve reasonable performance. With more complex prefetchers that fetch several consecutive addresses (+1, +2, +3...) bigger prefetcher capacity could give significantly better results.

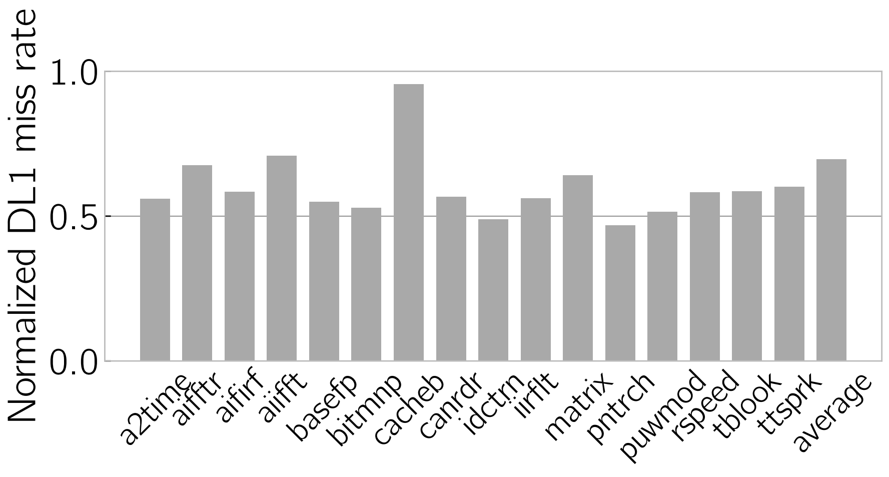


Figure 8.9: Normalized DL1 miss rate of the prefetcher

The prefetching in the L1 caches can be done for DL1, IL1 or both. On average, speedups at around 40% for each, DL1 and IL1, and around 55% for both together. Since DL1 caches have been regarded as particularly challenging due to their more irregular access patterns, and due to lack of space, we perform a detailed analysis only using prefetching on DL1.

Average execution time improvements with the prefetcher (Figure 8.8) are around 40% , with some benchmarks (idctrn, pntrch) showing more than a 60% speedup. For instance, for the 5 benchmarks that have more than a 50% speedup increase (`aifirf`, `idctrn`, `matrix`, `pntrch`, `puwmod`) their L2 miss rate goes from 85%-98% down to less than 50% in all them, reaching as low as 15% in `idctrn`, which is the one with the highest average speedup. In these benchmarks 30-40% of all instructions are memory operations (ld/st), which adds to the e significant decrease and high cost of L2 misses. For the two benchmarks that show a smaller average speedup increase (`a2time` and `bitmnp`), even though their decrease in L2 is significant, the total number of L2 accesses is low (in the order of thousands), limiting the impact of prefetching.

Regarding pWCET estimates, they are improved by 28% on average. Note that the particular variability of the execution times may cause pWCET curves to behave differently with and without prefetcher, thus leading to cases where pWCET gains are either lower or higher than average performance gains. Also as pWCET computation builds on those executions high execution times, benchmarks showing less pWCET improvement than average improvement correlate with those benchmarks wigh higher variations in the different execution runs with respect to the baseline results. In general, however, average and worst-case gains are similar in absolute terms but, since pWCET estimates are higher than average execution times, the relative gains tend to decrease.

In terms of DL1 miss rate, prefetchers reduce by 30% the total number misses (see Figure 8.9), and given the relative contribution of misses to execution time is high, this miss rate reduction has a high impact in execution time.

While the prefetcher naturally increases the total number of L2 accesses naturally

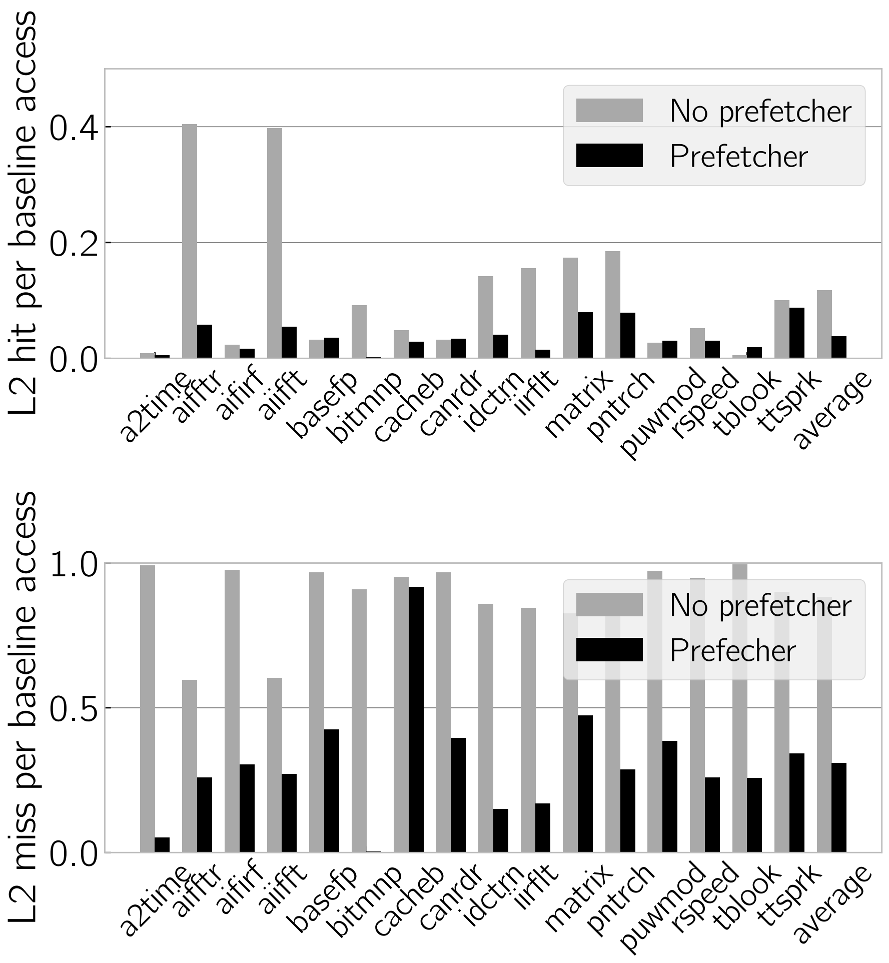


Figure 8.10: L2 hits/misses using as a baseline the L2 accesses without prefetcher

increases, prefetch accesses do not cause processor stalls since no particular instruction that could block the pipeline issued those requests. Moreover, contention in the access to L2 and memory caused by prefetch requests turns out to be tiny, as reflected in the performance results. Therefore, we study the impact of the prefetcher on on-demand L2 accesses caused by the program. In particular, Figure 8.10 shows, in two plots, program-triggered L2 hits and misses per access (hpa and mpa) normalized to the total number of L2 accesses produced without prefetcher. Note that:

$$hpa_{nopref} + mpa_{nopref} = 1$$

while this does not hold for the prefetcher and instead since DL1 experiences fewer misses thanks to the prefetcher.

$$hpa_{pref} + mpa_{pref} < 1$$

In the no prefetch setup, the L2 has mostly misses (almost 90%), since most of them are cold misses. In the prefetch setup, the number of hits is decreased from 12%

to 4%, but since hits take short latency and this number is really low, the impact on execution time is negligible. More interestingly, the 88% of misses that the L2 had drastically reduces down to 31% when the prefetch is active. Since L2 misses represent slow memory accesses, any reduction in these accesses will have a significant impact in execution time, as shown before.

8.6 Related Work

Software prefetching [37] adds prefetch instructions to the code that explicitly instruct the hardware to fetch data. However, the addition of these prefetch instructions not only affects the timing behavior of tasks, but also increase code size [102]. Despite these limitations, the use of software prefetching has been shown effective in the context of CRTES typically for managing scratchpads [139, 49, 144, 131].

Hardware prefetchers are commonly used in high performance systems [147, 44]. The most common hardware prefetchers are stream based [84] or stride based [18]. Other works propose methods for prefetching irregular access patterns [83]. Recent works in the literature focus on improving bandwidth efficiency [141], increasing accuracy [88] and improving the difficult-to-predict patterns [137].

When the Prefetchers used in embedded processors [14, 120] cannot be properly disabled, this creates uncertainty on the behaviors captured in measured execution times [55, 120].

Previous work has proposed the use of hardware prefetchers for Real-Time Systems [61, 12]. The study in [61] introduces prefetchers that improve average non-guaranteed execution time, guaranteeing the same WCET that would be had without prefetching. On the contrary, the prefetching schemes presented in this work are able to achieve performance improvements in WCET too. The authors of [12] propose a prefetcher for the instruction cache, building upon a model that uses ILP to compute WCET and cache parameters. The solution proposed in that paper is specific for the instruction cache, and cannot be easily used in the data cache.

8.7 Conclusions

The increasing need for higher performance in CRTES has led to the adoption of complex hardware features such as cache hierarchies and multicores. This work takes a step forward by enabling the use of hardware prefetchers to improve WCET estimates. In particular, our formal framework allows designing hardware prefetchers preserving time predictability by construction, which we show to deliver large performance gains, both on the average and worst-case, thus motivating the adoption of those features in future CRTES.

Chapter 9

Timing anomalies

9.1 Introduction

Dealing with timing anomalies, as explained in Section 2.1.5, is mandatory to enable and support the analysis of complex computing platforms. This applies to both Static Timing Analysis (STA), for which timing anomalies have been deeply analyzed [78, 158], and Measurement-Based Timing Analysis (MBTA) (MBTA) [155], for which instead timing anomalies have been totally neglected so far.

In this Chapter we promote a change of perspective, analyzing timing anomalies in the context of MBTA. This is particularly relevant for the automotive domain, where MBTA is the most widely used timing analysis technique. In particular, our contributions are:

1. We analyze timing anomalies in MBTA, concluding that anomalies cannot be handled as they are for STA. In particular, we observe that the challenges they bring to MBTA are not related to analysis (i.e. model) assumptions, but rather to the generic difficulty for the user to exercise sufficient control of all factors with bearing on timing during program runs in the analysis phase. Based on this observation, we tackle, for the first time, the challenge posed by timing anomalies on MBTA of high-performance processor designs, by analyzing their impact on Worst-Case Execution Time (WCET) estimates.
2. With emphasis on the probabilistic variant of MBTA, called MBPTA [5], we show how the use of those hardware features that can potentially cause timing anomalies can still be safely enabled in critical real-time systems. To that end, we leverage the use of time-randomized hardware designs (e.g. random placement and replacement caches, random arbitration in shared resources) that make the occurrence of timing anomalies probabilistic.
3. We instantiate the aforementioned key principles on a particular example of a type of timing anomaly as it may happen on a commercially available time-randomized processor design.

9.2 Timing Anomalies in MBTA

MBTA approaches have not considered timing anomalies as a concern so far, since the usual standpoint from which practitioners are used to consider timing anomalies is specifically that of STA. In our opinion, instead, some form of timing anomalies are to be considered relevant even in the scope of MBTA.

It is worth noting that, in the context of end-to-end program analysis, timing anomalies can happen due to hardware resources having variable latencies. Considering a sequence of instructions with fixed input data, accessing a fixed set of hardware resources, the sequence can lead to different execution times only in response to different initial processor states, inducing a variable response time (jitter) of a subset of the involved hardware components. While it is true that anomalies do not explicitly break any MBTA assumption, their potentially erratic impact on timing may jeopardize the fundamental conditions for measurement-based methods. For this reason, anomalies in MBTA need to be understood from the perspective of the *representativeness* of analysis-time observations.

Representativeness covers whether the measurements performed in the test campaigns during the *analysis phase* capture the worst-case relevant effects that can arise during system operation. Timing anomalies have the potential to distort the correspondence between analysis and operation conditions.

Observation 1. *With MBTA, dealing with timing anomalies builds on the ability to argue whether they have been triggered or not when running a program; whether they can actually occur during system operation; and whether their observed impact during analysis tests upperbounds the impact they may incur during operation.*

Figure 9.1, compares different ways in which timing anomalies – referred to as TA in the figure – can be attacked under the STA and MBTA paradigms. Similarly to STA, analyzing a system that can be proved to be timing-anomaly free would be the most favorable scenario. Unfortunately, assessing the lack of timing anomalies is not realistically affordable in the general case, and can only be possibly achieved with highly-specialized hardware designs [99].

Interestingly, modeling timing anomalies is not a challenge for MBTA. The challenge instead is to trigger anomalies during analysis tests and to size their impact, as they can manifest during operation. Theoretically, users are required to design experiments that capture the potential increase in execution time they entail.

Dealing with anomalies poses a challenge analogous to the one faced by end users to trigger specific low-level hardware interactions, since the user lacks explicit control knobs over them. To conclude the parallelism with STA, from the MBTA perspective, a relevant classification of timing anomalies does not focus on whether an anomaly has k-bounded versus domino effects but rather on *controllability*, which refers to whether or not an MBTA user is able to trigger them in a controlled way. Next, we look into timing anomalies from the perspective of MBTA, in the specific context of Measurement-Based Probabilistic Timing Analysis (MBPTA)-compliant hardware.

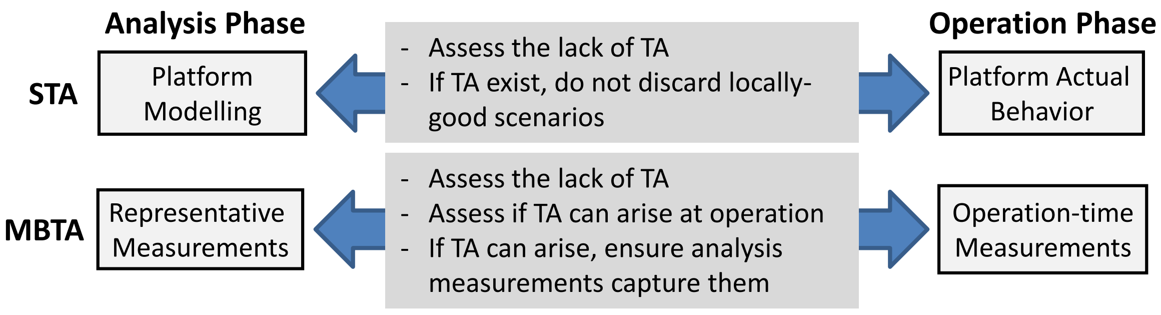

Analysis Phase
Operation Phase
STA
Platform Modelling
- Assess the lack of TA
- If TA exist, do not discard locally-good scenarios
Platform Actual Behavior
MBTA
Representative Measurements
- Assess the lack of TA
- Assess if TA can arise at operation
- If TA can arise, ensure analysis measurements capture them
Operation-time Measurements

Figure 9.1: STA and MBTA management of Timing Anomalies

9.3 Timing Anomalies in MBPTA

Timing anomalies in time-deterministic processors may potentially occur systematically. This is not an issue for the way STA deals with anomalies as the relevant aspect is whether an anomaly can either happen (and it is always assumed to) or not. Under MBTA, instead, the frequency at which an anomaly occurs has a variable impact on the execution time, which in turn could challenge the reliability of the WCET estimate. This concern is partially cured under MBPTA. In fact, for MBPTA, and in particular time-randomized hardware, certain timing events exhibit a random behavior, which can potentially break systematic patterns and allow building probabilistic arguments on the appearance of timing anomalies.

Observation 2. *Time-randomized processors (e.g., implementing caches [92] and buses [82] with time randomization properties), used in combination with MBPTA, trigger a number of random timing events that potentially break systematic timing behavior.*

As a result, the occurrence of some timing anomalies becomes inherently probabilistic. Moreover, the accumulation of timing anomalies occurs with decreasing probabilities. In fact, a given event potentially triggering an anomaly, necessarily repeats (chain of occurrence) with decreasing probabilities so that execution time variations end up being of lower magnitude than those produced by randomized hardware resources themselves. We will consolidate this argument while reasoning on a practical example in Section 9.4.

However, MBPTA and randomization do not prevent that some other timing anomalies may be systematically triggered, because they depend on non-time-randomized sources of execution time variation. Under some conditions, however, also these anomalies can be conveniently controlled.

9.3.1 Taxonomy of Timing Anomalies in MBPTA

In MBPTA-compliant processors, some individual sources of jitter (i.e. resources with variable latency) are controlled in a way that they are upper-bounded, whereas others – those with highest impact in WCET estimates – are time-randomized (e.g., cache memories, bus arbitration, etc.). These different sources of jitter have been analyzed in a LEON-like processor, currently assessed for future space missions, as part of the

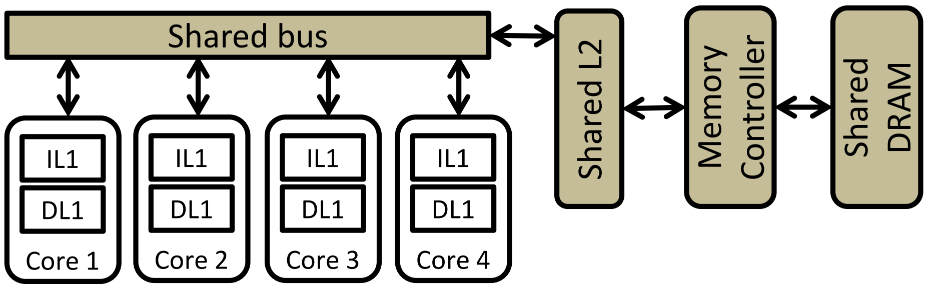


Figure 9.2: Processor architecture considered in this analysis. IL1, DL1 and L2 stands for first-level instruction and data caches; and L2 cache

PROXIMA Project [128].

Observation 3. *Constant execution time events cannot trigger any timing anomaly (but can propagate them).*

Those resources exhibiting a constant timing behavior exhibit the same behavior at analysis and operation, regardless of any execution condition. They cannot trigger any timing anomaly but cannot compensate nor prevent the effects of other anomalies potentially triggered.

Observation 4. *Random events whose execution time distribution does not change between analysis and operation, will exhibit probabilistically boundable timing anomalies.*

If the response time distribution of the resource remains unaltered between analysis and operation, then the occurrence and impact of timing anomalies can be related to the probability distribution observed at analysis. MBPTA is still responsible for guaranteeing representativeness of the observations.

From the observations above, we introduce a taxonomy of timing-anomaly scenarios for randomized architectures. Each hardware resource can be characterized as potentially triggering:

1. **No timing anomalies.** Fixed-latency timing events exhibit the same behavior at analysis and during operation. Hence, they trigger no timing anomaly. This classification applies to deterministic resources if the deterministic upper bound is enforced (by hardware means) even during operation.
2. **Probabilistically-controlled timing anomalies.** Some timing anomalies may be triggered by random events. Thus, they will be observed with a given probability in analysis runs. A rigorous MBPTA application [5] will guarantee that their timing impact is properly captured in the execution time measurements used to derive the Probabilistic Worst-Case Execution Time (pWCET) estimate, *as long as their probabilistic behavior is preserved from analysis time to operation.*
3. **Potentially uncontrolled timing anomalies.** Some timing anomalies may be triggered due to ***latent*** systematic effects, or due to probabilistic events whose probability distribution differs between analysis time and operation. Thus, their timing impact may not be properly captured in the execution time measurements used to feed MBPTA. As a result, the end user needs to account

for their effect without explicit support from the hardware or the timing analysis tool, which is a cumbersome task. In general, end users lack the degree of control needed to determine the impact of those anomalies and whether their impact has been properly accounted for, thus decreasing the quality of WCET estimates.

9.4 Dealing with Timing Anomalies

To illustrate how to deal with timing anomalies on an MBPTA-compliant hardware design, we use as example the platform described in Section 3.5. The main timing characteristics of such processor are illustrated in Figure 9.2 and further discussed in Section 9.5. We identify a number of sources of jitter – and so potential sources to trigger timing anomalies – in the processor design: `FDIV`/`FSQRT` operations, cache memories and randomly arbitrated resources. In the next sections we review the potential timing anomalies that could be triggered in such design, and discuss how jittery resources need to be controlled to avoid harmful (i.e., potentially uncontrolled) timing anomalies.

9.4.1 Upperbounding Variable-Latency Units

In our reference processor, `FDIV` and `FSQRT` incur jitter depending on the values operated. Following existing solutions for STA, we force these operations to take their worst latency [124], removing the jitter with minimum impact on average performance.

9.4.2 Priority Inversion in Cache Access

The requests sent to each cache (IL1, DL1 and L2) are served in arrival order. All requests to IL1 (DL1) are sent from the same pipeline stage, fetch (execution), and hence, the request arrival order matches the program order.

Let us assume instructions i_x and i_y are executed in program order, i.e., $x < y$, then all the requests these instructions can generate to IL1 ($i_{x,IL1}$, $i_{y,IL1}$) and DL1 ($d_{x,DL1}$, $d_{y,DL1}$) will be served in program order. However, this does not apply to L2, since the requests sent to L2 can be generated by instructions in two different stages (fetch and execute), which can generate inversion i.e. the instruction request of the second instruction $i_{y,L2}$ is served by the L2 before the data request of the first instruction $d_{x,L2}$. In case both requests to L2 are generated in the same cycle, $d_{x,L2}$ is prioritized.

When a memory request misses in a private L1, it needs to get access to the bus shared across the 4 cores to reach the L2 cache. Several MBPTA-compliant time-composable arbitration policies have been proposed [82], including round-robin policy. With this policy, we guarantee that, in a 4-core processor, each core will be able to access the shared L2 cache 1 out of every 4 time slots. Hence, the worst latency to

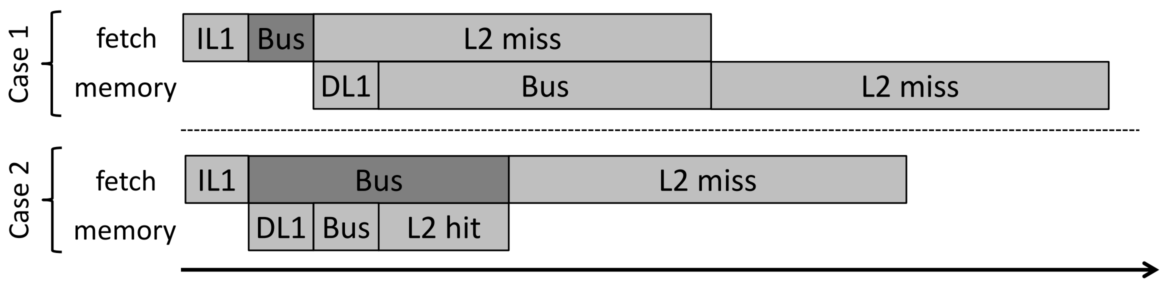


Figure 9.3: Priority inversion causing a timing anomaly

reach the L2 cache is 4 time slots minus 1 cycle (if requests can only be sent the first cycle of the slot).

Given this bus behavior, we can see in Figure 9.3 an example of a timing anomaly that occurs due to priority inversion. MBPTA-compliant time-composable arbitration policies impose accounting for worst-case contention during analysis, regardless of whether the arbitration policy is deterministic (e.g. round robin) or randomized (e.g. Random Permutations (RP) [82]). Either case, this makes that the delay experienced to access the bus may be typically high during analysis (case 2). Then, during operation, for efficiency reasons (e.g. average performance, power, etc.), worst-case contention is not enforced and requests experience *actual* contention, which will be typically lower, thus increasing the chances of experiencing timing anomalies (case 1).

In general, depending on the observed behaviours at analysis time (AT) and operation time (OT), we have two possible scenarios: (a) Priority inversion happens systematically at AT, or with the same (or higher) probability at AT than during OT. And (b) Priority inversion does not happen at AT, or occurs with lower probability than during OT. Scenario (a) is covered by MBPTA since execution time measurements during analysis account for worse conditions than those during operation [5]. In scenario (b), however, timing anomalies have not been accounted for properly, typically because their occurrence has been prevented (or heavily put down) as a side-effect of the analysis configurations and setups.

In scenario (b), the potential impact on timing that events not captured at AT can have later during OT needs to be understood and gauged. We do so by leveraging on the probabilistic nature of the enhanced LEON3 MBPTA-compliant platform, where random placement and replacement caches are used. In the particular case of the cache-access-inversion timing anomaly, we can reason on the probability of occurrence of its root causes: besides the bus delay, a cache miss in L2 – which has a fixed latency – is causing the eviction of a cache line that is accessed by an instruction sufficiently close in the pipeline. With random caches [92] the probability of an L2 miss to evict the useful cache line referenced by the DL1 miss is bounded by the number of cache sets (S_{L2}) and ways (W_{L2}) in L2, as shown in Equation 1 (P_{evict}). To upperbound the overall timing effect, we use the number of L2 misses potentially triggering the anomaly, which can be in general obtained by exploiting accurate hardware counters (e.g., L2_MISS_COUNT) as provided by the standard debug support unit (DSU). The total L2 miss count is a conservative overapproximation as it includes L2 misses that can never cause an anomaly by construction, or that were already triggering timing anomalies at AT. Equation 1 derives an upperbound to overall effects of cache-access-inversion anomaly (Δ) on a given program by collecting the cost of each potential anomaly, which is in turn bounded by the cost of the additional L2 cache miss.

$$\Delta \leq \underbrace{\frac{1}{S_{L2} \cdot W_{L2}}}_{P_{evict}} \times \texttt{L2\_MISS\_COUNT} \times \texttt{L2\_MISS\_LATENCY} \tag{9.1}$$

In the particular case of our target processor, the L2 cache is a 512KB 4-way 32B/line cache [71]. Moreover, since the L2 cache is partitioned across the 4 cores, each one receives exactly 1 cache way, which means that $P_{evict} \leq \frac{1}{4096 \cdot 1} \approx 0.000244$. The cost of an additional L2 cache miss is 28 cycles in the target platform.

The probability of occurrence of the anomaly (per-se objectively low), rapidly decreases when we consider for example the repeated execution of the same set of instructions within a loop since accesses in the following iterations will likely hit in IL1 and DL1. Looking at the specific anomaly, we also observe that both L2 accesses, instruction and data ones jointly contributing to the anomaly, must be initiated by instructions that can actually suffer from some form of inversion in the pipeline, which is only 7-stages in the LEON. As a result, the effect of an anomaly cannot propagate outside of its pipeline window. Borrowing the terminology used in the scope of static analysis, this anomaly can be classified as *boundable*.

9.4.3 Priority Inversion due to Initial Cache State

A similar priority inversion could occur due to the initial cache state. While worst-case initial state is enforced during analysis (e.g. empty write-through caches and useless dirty contents in write-back ones), some useful contents may be stored in cache during operation so that some accesses become hits. If those hits lead to timing anomalies, they do it with the same (very low) probability described before. Moreover, by turning misses into hits, execution time is lower and hence, less likely to be close to the WCET.

Equation 1 provides a parametric bound that depends on the number of misses generated by a program. In practice, however, the fact that the probability of occurrence of such anomalies is low and rapidly decreasing (when considering sequences of instructions) makes it arguable whether and to what extent they do actually pose a threat to the trustworthiness of pWCET bounds.

9.4.4 Managing Arbitration Effects on Requests

As shown in the previous section, the bus and memory controller arbitration policies can also impact timing anomalies. In particular, they determine the latency to serve L2 and memory requests and, indirectly, the order in which requests are served, which can generate a timing anomaly. However, these components process requests in order in the LEON processor, so they cannot produce further anomalies by themselves.

Table 9.1: Timing anomalies: identical distribution test results with timing anomalies (and 10X timing anomalies) w.r.t. no timing anomalies

	P-value timing anomalies	**P-value 10X timing anomalies**
`a2time`	1.0000	0.0009
`basefp`	1.0000	0.9987
`bitmnp`	1.0000	0.0196
`cacheb`	1.0000	0.0072
`canrdr`	1.0000	0.0010
`matrix`	1.0000	0.0776
`pntrch`	0.9995	0.0001
`puwmod`	1.0000	0.0546
`rspeed`	0.9689	0.0000
`tblook`	1.0000	0.9999
`ttsprk`	1.0000	0.9999
% PASSED	100%	44.4%

9.5 Evaluation

We use the Hardware Description Language (HDL) implementation of the NGMP processor described in Section 3.5, as well as the EEMBC [127] benchmarks (Section 3.7).

9.5.1 Impact of Timing Anomalies on Execution Time

In order to assess the impact of timing anomalies on execution time distributions, we perform the following experiment: from each execution time trace (i.e. execution time measurements for a given benchmark) we produce an additional execution time trace by decreasing each execution time by the expected upper-bound number of timing anomalies (IL1 misses divided by $W_{L2} \cdot S_{L2} = 4,096$) multiplied by the upper bound timing impact of a timing anomaly (28 cycles). The number of timing anomalies is obtained as the quotient of dividing the number of misses by $4,096$, and it is increased by 1 with a probability matching the remainder divided by $4,096$. For instance, if a program runs for 2,000,000 cycles and experiences 10,000 IL1 misses, the quotient of $\frac{10,000}{4,096}$ is 2 and the remainder $1,808$, so we decrease its execution time by 56 cycles with probability $\frac{4,096-1,808}{4,096}$ and by 84 cycles with probability $\frac{1,808}{4,096}$. Then we compare the those execution distributions against the original ones with the Kolmogorov-Smirnov two-sample identical distribution test with a significance level of $\alpha = 0.05$ [53]. This test returns a P-value in the range $[0, 1]$ that indicates that the identical distribution hypothesis cannot be rejected if the P-value is higher than 0.05.

The results of the test are shown in Table 9.1. As shown in the second column, even if the upper bound number of timing anomalies is applied with their upper bound timing impact, execution time distributions cannot be proven different. We have further multiplied the number of timing anomalies by a factor of 10, and then some benchmarks failed the test as shown in the third column. We have tried to correlate

Table 9.2: Timing anomalies: average execution time (cycles) and minimum, average and maximum number of IL1 misses

	Execution Time		**IL1 misses**		
	Average	**Stdev**	**Minimum**	**Average**	**Maximum**
`a2time`	334,854,930	918,841	626	30,336	543,753
`basefp`	944,861,123	41,471	506	955	59,143
`bitmnp`	870,384,078	387,726	452,891	527,722	636,748
`cacheb`	71,522,108	47,282	414	428	450
`canrdr`	132,303,886	115	332	338	353
`matrix`	1,940,292,855	55,143	71,048	74,097	89,117
`pntrch`	269,247,000	99	328	333	348
`puwmod`	87,946,616	224	409	421	436
`rspeed`	74,743,468	43	291	296	309
`tblook`	343,703,627	539,749	755	22,491	273,231
`ttsprk`	154,567,711	7,811	685	735	806

test pass/fail results with the IL1 miss distribution or the execution (see Table 9.2), but no clear correlation has been identified. This indicates that keeping the same execution time distribution (or not) directly depends on the particular distribution regardless of the execution time average or standard deviation. Still, in the case of the upper bounded timing anomalies (not increased by 10X) all programs obtain the same execution time distribution, so timing anomalies can be regarded as irrelevant. For the 10X case, since 10X timing anomalies lead to different distributions, this may affect the quality of the pWCET estimates. Next we analyze pWCET estimates in detail.

9.5.2 Impact of Timing Anomalies on pWCET

We have studied the impact that execution time distributions have on pWCET estimates. For that purpose we compute pWCET estimates at an exceedance threshold of 10^{-12} per run with MBPTA [5], although the same observations hold for other values. First, we have computed those pWCET estimates and computed confidence intervals for a significance level of $\alpha = 0.05$, thus meaning that the true pWCET value should fall in that interval with 95% confidence.

Since it is virtually impossible determining whether timing anomalies occurred as well as preventing or enforcing them in a real processor, we perform a statistical assessment of the potential impact of timing anomalies on execution time distributions. In particular, we perform the following experiment: from each execution time trace we produce an additional execution time trace by decreasing each execution time by the expected upper-bound number of timing anomalies (IL1 misses divided by $W_{L2} \cdot S_{L2} = 4,096$) multiplied by the upper bound timing impact of a timing anomaly (28 cycles). The number of timing anomalies is obtained as the quotient of dividing the number of misses by $4,096$, and it is increased by 1 with a probability matching the remainder divided by $4,096$. Then we compare those execution distributions against the original ones with the Kolmogorov-Smirnov two-sample identical

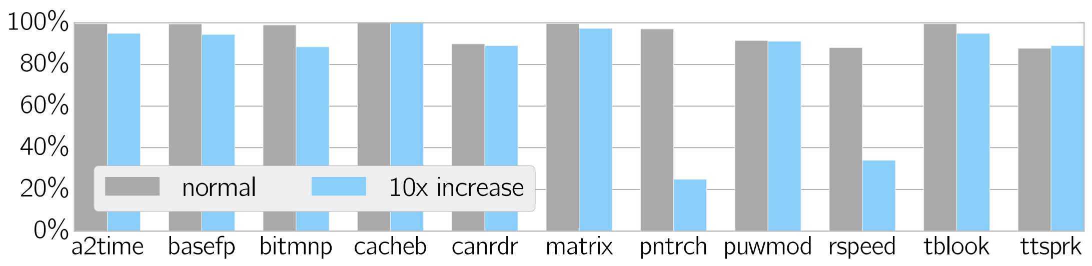


Figure 9.4: Normalized confidence interval overlap w/wo TA

distribution test with a significance level of $\alpha = 0.05$. This test, which returns a P-value in the range $[0, 1]$, indicates that the identical distribution hypothesis cannot be rejected when P-value > 0.05.

In order to compare pWCET estimates with and without timing anomalies, we do so in relative terms, by computing the ratio between the difference of both pWCET estimates and the actual pWCET estimate obtained without timing anomalies. Differences are completely negligible. The highest differences correspond to `a2time` and `tblook` (0.007% and 0.004% respectively), with most of them below 0.001%. We have further compared the confidence intervals to assess how much they overlap. This comparison is depicted in Figure 9.4 (left bar in each pair), where we can see that the overlap is huge, being always above 88% (95.5% on average) despite having very narrow confidence intervals in some cases. Therefore, pWCET estimates and execution time distributions cannot be proven different. Hence, the effect of timing anomalies on the overall timing is much lower than that already incurred by the variability of finite random samples.

We have repeated the very same analysis in the case where the upper-bounded impact of timing anomalies is further increased by a 10X factor. The difference between pWCET estimates with and without timing anomalies (with an impact increased by 10X) is very small. In particular, the difference is always below 0.1% and below 0.01% in most of the cases. In terms of confidence intervals, Figure 9.4 (right bar in each pair) shows that they overlap in all cases, so we cannot reject the hypothesis that both execution time distributions lead to the same pWCET estimate. However, we see that the relative overlap in the cases of `pntrch` and `rspeed` is comparatively low (25% and 34% respectively). This effect is produced by the extremely narrow confidence intervals (186 and 79 cycles respectively), so, in practice, pWCET estimates just differ by few tens of cycles.

9.6 Related Work

Different timing analysis strands deal with timing anomalies in the context of safety-related real-time systems [105, 78, 158, 30]. Among those works, an interesting classification is provided in [158], where processors are broken down into several categories depending on whether they are free of timing anomalies (ideal case), whether they have timing anomalies whose impact can be upper bounded with limited pessimism (good case), or whether they can trigger domino effects that might lead to high execution time impact (challenging case). Existing MBPTA-compliant processors are deemed as free of timing anomalies or have left their consideration for future work. This includes single [92] and multi-core [82, 153].

9.7 Conclusions

Timing anomalies can affect the quality of WCET estimates in the increasingly complex hardware used in critical real-time systems. We provide, for the first time, a definition of timing anomaly for MBTA that differs from that used in STA. We have also made an analysis of how to design hardware and collect measurements to limit – or even remove – the impact of certain timing anomalies for MB(P)TA. With an MBPTA-compliant RTL processor implemented in a FPGA, we assess the influence of timing anomalies on WCET estimates and show that their impact falls within the range of noise w.r.t. the own execution time variability.

Chapter 10

Conclusions

10.1 Contributions

The work carried out as part of this Thesis advances the state of the art of the critical real-time systems domain in several aspects. The main challenges tackled are the increase of guaranteed performance of multi-core processors, enabling of the use of reliability mechanisms on low level caches (those closest to the core) and the improvement of the confidence in Measurement-Based Probabilistic Timing Analysis (MBPTA).

These challenges have been tackled in the different contributions of the Thesis, which are the following:

- Our first contribution focuses on placement policies for multi-level caches. The solution sought takes into account the guaranteed performance, cost and complexity of the proposal and enables time composability in caches. Our proposal smartly combines existing placement policies so that composability is kept, which is key to reducing development cost and time, while preserving high performance and low complexity.
- Our second contribution focuses on MBPTA-compliant replacement policies that improve the guaranteed performance of the already existing Random Replacement (RR). We build on RR and Non-Most Recently Used (NMRU) with convenient properties such as limited pathological cases and preservation of data locality respectively. Moreover, we compare our policies with other policies like Least Recently Used (LRU) and Binary Tree (BT). We focus on two specific pathological scenarios that make some policies such as LRU and RR have bad performance. Our proposed solutions, Random Permutations (RP) and Non-Most Recently Used Random Permutations (NMRURP), do not suffer from such pathological scenarios while enabling MBPTA.
- Our third contribution focuses on reducing the contention in shared last-level caches. Our focus is to reduce average guaranteed execution times while reducing energy consumption and reducing the coherence messages and complexity. To that end, we propose a hybrid write technique that uses (1) write-back for

private data used by just one core, thus greatly reducing off-core accesses and hence, contention in the access to shared resources, and (2) write-through for data shared amongst cores, thus keeping the complexity to guarantee cache coherence low.

- We also focus on another important consideration for critical real-time systems, specially in some domains such as space and avionics where transient fault rates can be high due to radiation, thus requiring redundancy to mitigate potential errors. The use of write-back policies in low level caches (e.g. L1) challenges the cycle times of the memory pipeline stages due to the time needed to check-/generate Error Correction Codes (ECC). We consider several solutions to this problem, such as decreasing the frequency, pipelining ECC computation and the like, and propose adding an extra ECC stage and using a look-ahead technique to advance the computation of ECC to remove the cost of the extra stage most of the times. Our technique has small performance degradation w.r.t. unreliable designs while allowing the use of write-back in L1 caches.
- Our fifth contribution is a framework for enabling hardware prefetching in critical real-time systems. Hardware prefetching is commonly used in high-performance processors, but is not used in critical real-time systems since it is challenging to obtain timing guarantees. Our contribution is a formal framework that gives guarantees on the reduced execution times provided by a system with caches and a prefetcher, and provides the properties needed for a prefetcher to fit critical real-time systems requirements.
- The last contribution of this Thesis is about the definition, classification and management of timing anomalies in Measurement-Based Timing Analysis (MBTA). Timing anomalies, although studied extensively in Static Timing Analysis (STA), have not been studied in MBTA. We provide a definition of timing anomalies for MBTA as well as a classification of the types of timing anomalies for MBPTA-compliant processors. This first exploration of TA for MBTA is key to enable the use of these processors in a safe manner.

10.2 Impact

The design of high-performance multicore systems delivering timing guarantees is a real challenge that critical real-time industry faces when developing and bringing a product to the market. The work done in this Thesis eases this process in the following aspects:

The use of MBPTA can help lowering the costs of verification and validation of critical real-time systems, which are mandatory for products in this market. Since our proposed techniques are mostly based on MBPTA and focus on increasing the guaranteed and average performance delivered by processors for those systems, their addition to commercial processors should, in most cases, reduce the timing analysis costs and time when compared to solutions that use other Timing Analysis techniques, as well as increase confidence on the timing guarantees obtained.

The work done related to redundancy in Section 7 is a real challenge that Cobham Gaisler must address when designing the next generation of the LEON processor (LEON 5). While the actual design is proprietary and unknown to the general public, our proposal could serve as a starting point to solve this problem by preserving reliability levels with far lower performance cost than available solutions. Our proposal could also serve similar chip designs that are also challenged with this problem.

The contents of this Thesis relate mainly with two European Projects:

- **PROXIMA:** The aim of the PROXIMA project was to provide industry ready software timing analysis using probabilistic analysis for many-core and multi-core critical real-time embedded systems and to enable cost-effective verification of software timing analysis including worst case execution time. The work in this Thesis has been closely related with the findings obtained in the PROXIMA project. For instance, several placement and replacement policies proposed in PROXIMA (random placement, RR) have been used as the base to develop the contributions in Chapters 4, and 5. Furthermore, the setup used both in the simulator and in the Hardware Description Language (HDL) were developed in this project.
- **SuPerCom:** In the same line, the work on this Thesis has served as a basis for future techniques that will be explored in SuPerCom, an ERC consolidation grant that explores the combination of high-performance hardware and advanced statistics (including machine learning) for the critical-real time industry. This work will be done in the same research group where the Thesis has been developed, and as such, the techniques and proposals that have been developed in this Thesis will be in many cases the basis upon which to build new techniques in SuPerCom to improve guaranteed performance.

Last but not least, the work in this Thesis also contributes to answering key research challenges in the critical real-time domain, thus becoming the basis for future research:

- **Prefetchers:** Although software prefetching has been previously used in Critical Real-Time Embedded Systems (CRTES), there have been no proposals for a hardware prefetcher. This first step could enable multiple future research on the topic either improving the prefetching technique itself or iterating on the methodology used to enable the use of the prefetcher.
- **Timing anomalies:** Timing anomalies have been deeply studied in the context of STA, but this is the first work that defines and classifies timing anomalies in the context of MBTA. Future work could find other examples of timing anomalies in MBTA and propose how to tackle them, increasing the confidence in the Worst-Case Execution Time (WCET) estimates.

10.3 Future Work

The results and contributions of this Thesis can be further extended in several directions. We list some of these directions, presented in order of feasibility, from short term to long term.

Similar to the work done on replacement policies, still several already existing high-performance techniques are unexplored. For instance, other replacement policies for last-level shared caches such as RRIP [79] or BRRIP [79] could be adapted to be MBPTA compliant, thus improving guaranteed performance. Work related to this area will be more critical with future multi-cores that will increase the core count, making shared last-level caches' performance crucial for overall system performance.

Related to the previous point, contention in last-level caches is going to be a more relevant challenge with the increase of core counts. More intelligent coherence techniques that minimize the movement of data across the memory hierarchy will also be crucial to performance. In the same line of the previous research direction, more advanced write policies and coherence policies for complex NoCs should be investigated, such as limiting the timing impact of coherence management of mesh-based many-cores. While adapting write and coherence policies to be efficient and safe in critical real-time systems can be a hard task, the potential improvement makes it an interesting topic to research.

In terms of hardware prefetching, the prefetching technique itself that was used in this Thesis was relatively simple, since the main challenge was to be able to enable prefetching. Abundant solutions on high-performance prefetchers could be adapted to fit in a more relaxed version of our proposed framework. Future work could improve the prefetcher proposed in this Thesis to provide higher guaranteed performance with limited complexity and costs.

An even more challenging research area are systems comprising CPU and GPU on the same die (System on a Chip). These setups are becoming increasingly popular due to the performance and power needs of industries such as the automotive one. Autonomous driving is gaining importance in both the academic and industrial communities due to its potential impact on people's lives. However, several of its machine learning algorithms (for instance for object detection) require huge computing resources that can be satisfied by GPUs. Thus, these systems will be key in bringing these features to consumers. However, the use of GPUs in critical real-time systems is in its infancy. The difference in the computing paradigm of CPUs vs GPUs, more focused on throughput than latency, can make it challenging to obtain timing guarantees. Several techniques developed in this Thesis could be used as a starting point for GPUs, including at least write policies, prefetch schemes and management of timing anomalies.

Bibliography

[1] (2012). *ARINC Specification 653: Avionics Application Software Standard Standard Interface, Part 1 and 4*. ARINC.

[2] (2015). ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. `https://www.arm.com/company/news/2015/04/arm-expects-vehicle-compute-performance-to-increase-100x-in-next-decade`, ARM Press Release.

[3] Abella, J., Carretero, J., Chaparro, P., Vera, X., and González, A. (2009). Low vccmin fault-tolerant cache with highly predictable performance. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 111–121.

[4] Abella, J., Hernandez, C., Quiñones, E., Cazorla, F. J., Conmy, P. R., Azkarate-Askasua, M., Perez, P., Mezzetti, E., and Vardanega, T. (2015). Wcet analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 1–10.

[5] Abella, J., Padilla, M., Castillo, J. D., and Cazorla, F. J. (2017). Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM Trans. Des. Autom. Electron. Syst.*, 22(4).

[6] Abella, J., Quiñones, E., Cazorla, F. J., Sazeides, Y., and Valero, M. (2011). Rvc: A mechanism for time-analyzable real-time processors with faulty caches. In *HiPEAC '11*, page 97–106, New York, NY, USA. Association for Computing Machinery.

[7] Abella, J., Quiñones, E., Wartel, F., Vardanega, T., and Cazorla, F. J. (2014). Heart of gold: Making the improbable happen to increase confidence in mbpta. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 255–265.

[8] Agah, A., Fakhraie, S. M., and Emami-Neyestanak, A. (2007). Tertiary-tree 12-ghz 32-bit adder in 65nm technology. In *2007 IEEE International Symposium on Circuits and Systems (ISCAS)*, pages 3006–3009.

[9] Agirre, I., Azkarate-Askasua, M., Hernandez, C., Abella, J., Perez, J., Vardanega, T., and 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*, pages 677–684.

[10] Al-Zoubi, H., Milenkovic, A., and Milenkovic, M. (2004). Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *Proceedings of the 42nd Annual Southeast Regional Conference*, ACM-SE 42, page 267–272, New York, NY, USA. Association for Computing Machinery.

[11] Altera. Altera quartus ii. `https://www.altera.com/downloads/download-center.html`.

[12] Aparicio, L. C., Segarra, J., Rodríguez, C., and Viñals, C. (2010). Combining prefetch with instruction cache locking in multitasking real-time systems. In *2010 IEEE 16th International Conference on Embedded and Real-Time Computing Systems and Applications*, pages 319–328.

[13] Araujo, B. A., Gracioli, G., Kloda, T., Hoornaert, D., and Caccamo, M. (2021). Implementation and evaluation of adaptive cache insertion policies for real-time systems. In *2021 XI Brazilian Symposium on Computing Systems Engineering (SBESC)*, pages 1–8.

[14] ARM. Arm cortex-a72 mpcore processor.

[15] ARM. ARM Cortex-M7 processor. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0489b/DDI0489B_cortex_m7_trm.pdf`.

[16] ARM. ARM Cortex R5 technical reference manual. `http://infocenter.arm.com/help/topic/com.arm.doc.ddi0460d/DDI0460D_cortex_r5_r1p2_trm.pdf`.

[17] ARM (2006). *Cortex-R4 and Cortex-R4F Technical Reference Manual.*

[18] Baer, J. L. and Chen, T. F. (1991). An effective on-chip preloading scheme to reduce data access penalty. In *Supercomputing '91:Proceedings of the 1991 ACM/IEEE Conference on Supercomputing*, pages 176–186.

[19] Bartolini, S., Foglia, P., and Prete, C. A. (2018). Exploring the relationship between architectures and management policies in the design of nuca-based chip multicore systems. In *Future Generation Computer Systems.*

[20] Belady, L. A. (1966). A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101.

[21] Benedicte, P., Abella, J., Hernandez, C., Mezzetti, E., and Cazorla, F. J. (2019a). Towards limiting the impact of timing anomalies in complex real-time processors. In *Proceedings of the 24th Asia and South Pacific Design Automation Conference*, ASPDAC '19, page 27–32, New York, NY, USA. Association for Computing Machinery.

[22] Benedicte, P., Hernandez, C., Abella, J., and Cazorla, F. J. (2018a). Design and integration of hierarchical-placement multi-level caches for real-time systems. In *2018 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 455–460.

[23] Benedicte, P., Hernandez, C., Abella, J., and Cazorla, F. J. (2018b). HWP: Hardware Support to Reconcile Cache Energy, Complexity, Performance and WCET Estimates in Multicore Real-Time Systems. In Altmeyer, S., editor, *30th Euromicro Conference on Real-Time Systems (ECRTS 2018)*, volume 106 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 3:1–3:22, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[24] Benedicte, P., Hernandez, C., Abella, J., and Cazorla, F. J. (2018c). Rpr: A random replacement policy with limited pathological replacements. In *Proceedings of the 33rd Annual ACM Symposium on Applied Computing*, SAC '18, page 593–600, New York, NY, USA. Association for Computing Machinery.

[25] Benedicte, P., Hernandez, C., Abella, J., and Cazorla, F. J. (2019b). Laec: Look-ahead error correction codes in embedded processors l1 data cache. In *2019 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 818–823.

[26] Benedicte, P., Hernandez, C., Abella, J., and Cazorla, F. J. (2019c). Locality-aware cache random replacement policies. In *Journal of Systems Architecture.*

[27] Benedicte, P., Hernández, C., Abella, J., and Cazorla, F. J. (2018d). Book of abstracts, 5th bsc severo ochoa doctoral symposium. In *Improving Time-Randomized Cache Design.*

[28] Benedicte, P., Kosmidis, L., Quinones, E., Abella, J., and Cazorla, F. J. (2016a). Modelling the confidence of timing analysis for time randomised caches. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, pages 1–8.

[29] Benedicte, P., Kosmidis, L., Quiñones, E., Abella, J., and Cazorla, F. J. (2016b). A confidence assessment of wcet estimates for software time randomized caches. In *2016 IEEE 14th International Conference on Industrial Informatics (INDIN)*, pages 90–97.

[30] Binder, B., Asavoae, M., Brandner, F., Ben Hedia, B., and Jan, M. (2020). Scalable detection of amplification timing anomalies for the superscalar tricore architecture. In ter Beek, M. H. and Ničković, D., editors, *Formal Methods for Industrial Critical Systems*, pages 151–169, Cham. Springer International Publishing.

[31] Blaß, T., Hahn, S., and Reineke, J. (2017). Write-Back Caches in WCET Analysis. In Bertogna, M., editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 26:1–26:22, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[32] Bodin, F. and Seznec, A. (1997). Skewed associativity improves program performance and enhances predictability. *IEEE Transactions on Computers*, 46(5):530–544.

[33] Bowman, K. A., Duvall, S. G., and Meindl, J. D. (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*, 37(2):183–190.

[34] Box, G. E. P. and Pierce, D. A. (1970). Distribution of residual autocorrelations in autoregressive-integrated moving average time series models. *Journal of the American Statistical Association*, 65:1509–1526.

[35] Buttle, D. (2012). ETAS GmbH, germany, real-time in the prime-time. keynote talk. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

[36] C., F. and R., W. (1999). Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems Journal*, 17:131–181.

[37] Callahan, D., Kennedy, K., and Porterfield, A. (1991). Software prefetching. In *Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS IV, page 40–52, New York, NY, USA. Association for Computing Machinery.

[38] Cazorla, F. J., Abella, J., Andersson, J., Vardanega, T., Vatrinet, F., Bate, I., Broster, I., Azkarate-Askasua, M., Wartel, F., Cucu, L., Cros, F., Farrall, G., Gogonel, A., Gianarro, A., Triquet, B., Hernandez, C., Lo, C., Maxim, C., Morales, D., Quinones, E., Mezzetti, E., Kosmidis, L., Aguirre, I., Fernandez, M., Slijepcevic, M., Conmy, P., and Talaboulma, W. (2016). Proxima: Improving measurement-based timing analysis through randomisation and probabilistic analysis. In *2016 Euromicro Conference on Digital System Design (DSD)*, pages 276–285.

[39] Cazorla, F. J., Kosmidis, L., Mezzetti, E., Hernandez, C., Abella, J., and Vardanega, T. (2019). Probabilistic worst-case timing analysis: Taxonomy and comprehensive survey. *ACM Comput. Surv.*, 52(1).

[40] Charette, R. N. (2009). This car runs on code. *IEEE Spectrum*.

[41] Chaudhuri, M. (2009). Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches. In *2009 42nd Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 401–412.

[42] Chen, C. L. and Hsiao, M. Y. (1984). Error-correcting codes for semiconductor memory applications: A state-of-the-art review. *IBM Journal of Research and Development*, 28(2):124–134.

[43] Cobham Gaisler. Flight Software Workshop 2017. `http://flightsoftware.jhuapl.edu/files/2017/Day-3/02-Hellstrom-Cobham-HiRel.pdf`.

[44] Corporation, I. Intel64 and ia-32 architectures optimization reference manual.

[45] Cucu-Grosjean, L., Santinelli, L., Houston, M., Lo, C., Vardanega, T., Kosmidis, L., Abella, J., Mezzetti, E., Quiñones, E., and Cazorla, F. J. (2012). Measurement-based probabilistic timing analysis for multi-path programs. In *2012 24th Euromicro Conference on Real-Time Systems*, pages 91–101.

[46] Cuesta, B., Ros, A., Gómez, M. E., Robles, A., and Duato, J. (2011). Increasing the effectiveness of directory caches by deactivating coherence for private memory blocks. In *2011 38th Annual International Symposium on Computer Architecture (ISCA)*, pages 93–103.

[47] Dasari, D., Andersson, B., Nelis, V., Petters, S. M., Easwaran, A., and Lee, J. (2011). Response time analysis of COTS-based multicores considering the contention on the shared memory bus. In *2011 IEEE 10th International Conference on Trust, Security and Privacy in Computing and Communications*, pages 1068–1075.

[48] Davis, R. I. and Cucu-Grosjean, L. (2019). A survey of probabilistic timing analysis techniques for real-time systems. *Leibniz Transactions on Embedded Systems*, 6(1):03:1–03:60.

[49] Deverge, J. F. and I., P. (2007). WCET-directed dynamic scratchpad memory allocation of data. In *19th Euromicro Conference on Real-Time Systems (ECRTS'07)*, pages 179–190.

[50] Díaz, E., Abella, J., Mezzetti, E., Agirre, I., Azkarate-askasua, M., Vardanega, T., and Cazorla, F. J. (2016). Mitigating software-instrumentation cache effects in measurement-based timing analysis. In Schoeberl, M., editor, *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, volume 55 of *OASICS*, pages 1:1–1:11. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[51] Díaz, E., Fernández, M., Kosmidis, L., Mezzetti, E., Hernandez, C., Abella, J., and Cazorla, F. J. (2017). Mc2: Multicore and cache analysis via deterministic and probabilistic jitter bounding. In Blieberger, J. and Bader, M., editors, *Reliable Software Technologies – Ada-Europe 2017*, pages 102–118, Cham. Springer International Publishing.

[52] Duwe, H., Jian, X., and Kumar, R. (2015). Correction prediction: Reducing error correction latency for on-chip memories. In *2015 IEEE 21st International Symposium on High Performance Computer Architecture (HPCA)*, pages 463–475.

[53] Feller, W. (1996). *An Introduction to Probability Theory and Its Applications*. Wiley Series in Probability and Statistics. Wiley.

[54] Fernandez, G., Abella, J., Quiñones, E., Rochange, C., Vardanega, T., and Cazorla, F. J. (2014). Contention in Multicore Hardware Shared Resources: Understanding of the State of the Art. In Falk, H., editor, *14th International Workshop on Worst-Case Execution Time Analysis*, volume 39 of *OpenAccess Series in Informatics (OASIcs)*, pages 31–42, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[55] Fernandez, G., Cazorla, F. J., and Abella, J. (2018). Consumer electronics processors for critical real-time systems: a (failed) practical experience. In *9th*

European Congress on Embedded Real Time Software and Systems (ERTS 2018), Toulouse, France.

[56] Fernández, M., Gioiosa, R., Quiñones, E., Fossati, L., Zulianello, M., and 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, page 175–184, New York, NY, USA. Association for Computing Machinery.

[57] Fernandez, M., Morales, D., Kosmidis, L., Bardizbanyan, A., Broster, I., Hernandez, C., Quinones, E., Abella, J., Cazorla, F. J., Machado, P., and Fossati, L. (2017). Probabilistic timing analysis on time-randomized platforms for the space domain. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, page 738–739, Leuven, BEL. European Design and Automation Association.

[58] Gaisler, C. *LEON4-N2X Data Sheet and User's Manual.*

[59] Gaisler, C. NGMP preliminary datasheet version 2.1. `http://microelectronics.esa.int/gr740/LEON4-NGMP-DRAFT-2-1.pdf`.

[60] Gaisler, C. UT699 32-bit fault-tolerant SPARC V8/LEON 3FT processor data sheet. `http://www.gaisler.com/doc/gr712rc-datasheet.pdf`.

[61] Garside, J. and Audsley, N. C. (2014). Wcet preserving hardware prefetch for many-core real-time systems. In *Proceedings of the 22nd International Conference on Real-Time Networks and Systems*, RTNS '14, page 193–202, New York, NY, USA. Association for Computing Machinery.

[62] GbR, A. (2006). *Technical Overview.* AUTOSAR GbR, 2.0.1 edition.

[63] Girbal, S., Moretó, M., Grasset, A., Abella, J., Quiñones, E., Cazorla, F. J., and Yehia, S. (2013). On the convergence of mainstream and mission-critical markets. In *The 50th Annual Design Automation Conference 2013, DAC '13, Austin, TX, USA, May 29 - June 07, 2013*, pages 185:1–185:10. ACM.

[64] Gustafsson, J., Betts, A., Ermedahl, A., and Lisper, B. (2010). The Mälardalen WCET Benchmarks: Past, Present And Future. In Lisper, B., editor, *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, volume 15 of *OpenAccess Series in Informatics (OASIcs)*, pages 136–146, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik. The printed version of the WCET'10 proceedings are published by OCG (www.ocg.at) - ISBN 978-3-85403-268-7.

[65] Hahn, S., Reineke, J., and Wilhelm, R. (2015). Towards compositionality in execution time analysis: Definition and challenges. In *SIGBED Rev.*, volume 12, page 28–36, New York, NY, USA. Association for Computing Machinery.

[66] Hamming, R. W. (1950). Error detecting and error correcting codes. *The Bell System Technical Journal*, 29(2):147–160.

[67] Hardavellas, N., Ferdman, M., Falsafi, B., and Ailamaki, A. (2009). Reactive nuca: Near-optimal block placement and replication in distributed caches. In *Proceedings of the 36th Annual International Symposium on Computer Architecture*, ISCA '09, page 184–195, New York, NY, USA. Association for Computing Machinery.

[68] Hardy, D., Piquet, T., and Puaut, I. (2009). Using bypass to tighten wcet estimates for multi-core processors with shared instruction caches. In *2009 30th IEEE Real-Time Systems Symposium*, pages 68–77.

[69] Hennessy, J. and Patterson, D. (2017). *Computer Architecture: A Quantitative Aproach - 6th Edition.* Morgan Kaufmann.

[70] Hernandez, C., Abella, J., Cazorla, F. J., Andersson, J., and Gianarro, A. (2015). Towards making a LEON3 multicore compatible with probabilistic timing analysis. In *DASIA*.

[71] Hernández, C., Abella, J., Cazorla, F. J., Bardizbanyan, A., Andersson, J., Cros, F., and Wartel, F. (2017). Design and Implementation of a Time Predictable Processor: Evaluation With a Space Case Study. In Bertogna, M., editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 16:1–16:23, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[72] Hernandez, C., Abella, J., Gianarro, A., Andersson, J., and Cazorla, F. J. (2016). Random modulo: A new processor cache design for real-time critical systems. In *2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC)*, pages 1–6.

[73] Ho, N., Ashraf, I. I., Kaufmann, P., and Platzner, M. (2017). Accurate private/shared classification of memory accesses: A run-time analysis system for the leon3 multi-core processor. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '17, page 788–793, Leuven, BEL. European Design and Automation Association.

[74] Hsiao, M. Y. (1970). A class of optimal minimum odd-weight-column SEC-DED codes. *IBM Journal of Research and Development*, 14(4):395–401.

[75] IBM (2008). *PowerPC 750GX Lockstep Facility. Application note.* IBM.

[76] Infineon (2012). AURIX - TriCore datasheet.

[77] International Organization for Standardization (2009). *ISO/DIS 26262. Road Vehicles – Functional Safety.*

[78] J., R., B., W., S., T., R., W., I., P., J., E., and B., B. (2006). A Definition and Classification of Timing Anomalies. In Mueller, F., editor, *6th International Workshop on Worst-Case Execution Time Analysis (WCET'06)*, volume 4 of *OpenAccess Series in Informatics (OASIcs)*, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[79] Jaleel, A., Theobald, K. B., Steely, S. C., and Emer, J. (2010). High performance cache replacement using re-reference interval prediction (rrip). In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 60–71, New York, NY, USA. Association for Computing Machinery.

[80] Jalle, J., Abella, J., Fossati, L., Zulianello, M., and Cazorla, F. J. (2016a). Validating a timing simulator for the ngmp multicore processor. In Ouwehand, L., editor, *DASIA 2016 - Data Systems In Aerospace*, volume 736 of *ESA Special Publication*, page 7.

[81] Jalle, J., Fernandez, M., Abella, J., Andersson, J., Patte, M., Fossati, L., Zulianello, M., and Cazorla, F. J. (2016b). Bounding Resource Contention Interference in the Next-Generation Microprocessor (NGMP). In *8th European Congress on Embedded Real Time Software and Systems (ERTS 2016)*, TOULOUSE, France.

[82] Jalle, J., Kosmidis, L., Abella, J., Quiñones, E., and Cazorla, F. J. (2014). Bus designs for time-probabilistic multicore processors. In *2014 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 1–6.

[83] Joseph, D. and Grunwald, D. (1999). Prefetching using markov predictors. *IEEE Transactions on Computers*, 48(2):121–133.

[84] Jouppi, N. (1990). Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *[1990] Proceedings. The 17th Annual International Symposium on Computer Architecture*, pages 364–373.

[85] Karedla, R., Love, J., and Wherry, B. (1994). Caching strategies to improve disk system performance. *Computer*, 27(3):38–46.

[86] Kaushik, A. M., Hassan, M., and Patel, H. (2021). Designing predictable cache coherence protocols for multi-core real-time systems. *IEEE Transactions on Computers*, 70(12):2098–2111.

[87] Kim, H., de Niz, D., Andersson, B., Klein, M., Mutlu, O., and Rajkumar, R. (2014). Bounding memory interference delay in cots-based multi-core systems. In *2014 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 145–154.

[88] Kim, J., Pugsley, S. H., Gratz, P. V., Reddy, A. N., Wilkerson, C., and Chishti, Z. (2016). Path confidence based lookahead prefetching. In *2016 49th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 1–12.

[89] Koh, C., Wong, W., Chen, Y., and Li, H. (2009). Tolerating process variations in large, set-associative caches: The buddy cache. *ACM Trans. Archit. Code Optim.*, 6(2).

[90] Kopetz, H. (1997). *Real-Time Systems: Design Principles for Distributed Embedded Applications*. Real-Time Systems Series. Springer.

[91] Koren, I. and Koren, Z. (1998). Defect tolerance in VLSI circuits: techniques and yield analysis. *Proceedings of the IEEE*, 86(9):1819–1838.

[92] Kosmidis, L., Abella, J., Quiñones, E., and Cazorla, F. J. (2013a). A cache design for probabilistically analysable real-time systems. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 513–518.

[93] Kosmidis, L., Abella, J., Quiñones, E., and Cazorla, F. J. (2013b). Multi-level unified caches for probabilistically time analysable real-time systems. In *2013 IEEE 34th Real-Time Systems Symposium*, pages 360–371.

[94] Kosmidis, L., Abella, J., Wartel, F., Quiñones, E., Colin, A., and Cazorla, F. J. (2014). PUB: Path upper-bounding for measurement-based probabilistic timing analysis. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 276–287.

[95] Kosmidis, L., Quiñones, E., Abella, J., Vardanega, T., and Cazorla, F. J. (2013c). Achieving timing composability with measurement-based probabilistic timing analysis. In *16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013)*, pages 1–8.

[96] Kosmidis, L., Vardanega, T., Abella, J., Quiñones, E., and Cazorla, F. J. (2013d). Applying Measurement-Based Probabilistic Timing Analysis to Buffer Resources. *13th International Workshop on Worst-Case Execution Time Analysis*, 30:97–108.

[97] Kędzierski, K., Moreto, M., Cazorla, F. J., and Valero, M. (2010). Adapting cache partitioning algorithms to pseudo-lru replacement policies. In *2010 IEEE International Symposium on Parallel Distributed Processing (IPDPS)*, pages 1–12.

[98] Lahiri, K., Raghunathan, A., and Lakshminarayana, G. (2001). LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs. In *Proceedings of the 38th Design Automation Conference*, pages 15–20.

[99] Law, S. and Bate, I. (2016). Achieving appropriate test coverage for reliable measurement-based timing analysis. In *Euromicro Conference on Real-Time Systems (ECRTS)*.

[100] Lebeck, A. and Wood, D. (1994). Cache profiling and the SPEC benchmarks: a case study. *Computer*, 27(10):15–26.

[101] Lee, C., Potkonjak, M., and Mangione-Smith, W. (1997). MediaBench: a tool for evaluating and synthesizing multimedia and communications systems. In *Proceedings of 30th Annual International Symposium on Microarchitecture*, pages 330–335.

[102] Lee, J., Kim, H., and Vuduc, R. (2012). When prefetching works, when it doesn't, and why. *ACM Trans. Archit. Code Optim.*, 9(1).

[103] Lesage, B., Hardy, D., and Puaut, I. (2010). Shared data caches conflicts reduction for WCET computation in multi-core architectures. In *18th International Conference on Real-Time and Network Systems.*

[104] Li, Y., Suhendra, V., Liang, Y., Mitra, T., and Roychoudhury, A. (2009). Timing analysis of concurrent programs running on shared cache multi-cores. In *2009 30th IEEE Real-Time Systems Symposium*, pages 57–67.

[105] Lundqvist, T. and Stenstrom, P. (1999). Timing anomalies in dynamically scheduled microprocessors. In *Proceedings 20th IEEE Real-Time Systems Symposium (Cat. No.99CB37054)*, pages 12–21.

[106] Maric, B., Abella, J., Cazorla, F. J., and Valero, M. (2014a). Hybrid cache designs for reliable hybrid high and ultra-low voltage operation. *ACM Trans. Des. Autom. Electron. Syst.*, 20(1).

[107] Maric, B., Abella, J., and Valero, M. (2013). Efficient cache architectures for reliable hybrid voltage operation using edc codes. In *2013 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 917–920.

[108] Maric, B., Abella, J., and Valero, M. (2014b). Analyzing the efficiency of l1 caches for reliable hybrid-voltage operation using edc codes. *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, 22(10):2211–2215.

[109] McNairy, C. and Mayfield, J. (2005). Montecito error protection and mitigation. In *HPCRI'05: 1st Workshop on High Performance Computing Reliability Issues, in conjunction with HPCA'05.*

[110] Mezzetti, E., Fernandez, M., Bardizbanyan, A., Agirre, I., Abella, J., Vardanega, T., and Cazorla, F. J. (2017). Epc enacted: Integration in an industrial toolbox and use against a railway application. In *2017 IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 163–174.

[111] Mezzetti, E. and Vardanega, T. (2011). On the industrial fitness of WCET analysis. In *Workshop on Worst-Case Execution Time Analysis.*

[112] Mezzetti, E. and Vardanega, T. (2013). A rapid cache-aware procedure positioning optimization to favor incremental development. In *2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS)*, pages 107–116.

[113] Michael Barr. Real men program in C. `https://www.embedded.com/electronics-blogs/barr-code/4027479/Real-men-program-in-C`.

[114] Moseley, T., Kihm, J., Connors, D., and Grunwald, D. (2005). Methods for modeling resource contention on simultaneous multithreading processors. In *2005 International Conference on Computer Design*, pages 373–380.

[115] Muralimanohar, N., Balasubramonian, R., and Jouppi, N. P. (2009). CACTI 6.0: A tool to understand large caches. In *HP Tech Report HPL-2009-85*.

[116] Normand, E. (1996). Single event upset at ground level. *IEEE Transactions on Nuclear Science*, 43(6):2742–2750.

[117] Nowotsch, J., Paulitsch, M., Bühler, D., Theiling, H., Wegener, S., and Schmidt, M. (2014). Multi-core interference-sensitive wcet analysis leveraging runtime resource capacity enforcement. In *2014 26th Euromicro Conference on Real-Time Systems*, pages 109–118.

[118] NXP. MPC8245 integrated processor hardware specifications. `https://www.nxp.com/docs/en/data-sheet/MPC8245EC.pdf`.

[119] NXP (2019a). QorIQ P4080 Data Sheet. In *NXP*.

[120] NXP (2019b). QorIQ T2080 Data Sheet. In *NXP*.

[121] Owens, J. (2015). Delphi automotive, the design of innovation that drives tomorrow. Keynote talk.

[122] P., A. (1996). *Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators*. Xilinx.

[123] P., C. and R., C. (2004). *Building the Information Society: Basic Concepts of Abstract Interpretation*. IFIP International Federation for Information Processing. Springer.

[124] Paolieri, M., Quiñones, E., Cazorla, F. J., Bernat, G., and 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 '09, page 57–68, New York, NY, USA. Association for Computing Machinery.

[125] Paolieri, M., Quiñones, E., Cazorla, F. J., and Valero, M. (2009b). An analyzable memory controller for hard real-time CMPs. *IEEE Embedded Systems Letters*, 1(4):86–90.

[126] Pérez-Cerrolaza, J., Obermaisser, R., Abella, J., Cazorla, F. J., Grüttner, K., Agirre, I., Ahmadian, H., and Allende, I. (2020). Multi-core devices for safety-critical systems: A survey. *ACM Comput. Surv.*, 53(4):79:1–79:38.

[127] Poovey, J. A., Conte, T. M., Levy, M., and Gal-On, S. (2009). A benchmark characterization of the EEMBC benchmark suite. *IEEE Micro*, 29(5):18–29.

[128] PROXIMA (2014). Probabilistic real-time control of mixed-criticality multicore and manycore systems.

[129] Qureshi, M. K., Jaleel, A., Patt, Y. N., Steely, S. C., and Emer, J. (2007). Adaptive insertion policies for high performance caching. In *Proceedings of the 34th Annual International Symposium on Computer Architecture*, ISCA '07, page 381–391, New York, NY, USA. Association for Computing Machinery.

[130] Ros, A. and Kaxiras, S. (2012). Complexity-effective multicore coherence. In *Proceedings of the 21st International Conference on Parallel Architectures and Compilation Techniques*, PACT '12, page 241–252, New York, NY, USA. Association for Computing Machinery.

[131] S., K. (2001). Using scratchpad memory for stack data in hard real-time embedded systems. In *Proceedings of the Memory Architecture and Organization Workshop*.

[132] S., K. and S., N. (2000). *Extreme value distributions: theory and applications*. World Scientific.

[133] Schlansker, M. S., Shaw, R., and S., S. (1993). Randomization and associativity in the design of placement-insensitive caches. *HP Tech Report*, HPL-93-41.

[134] Schoeberl, M. (2009). Time-predictable cache organization. In *2009 Software Technologies for Future Dependable Distributed Systems*, pages 11–16.

[135] Scolari, A., Bartolini, D. B., and Santambrogio, M. D. (2016). A software cache partitioning system for hash-based caches. volume 13, New York, NY, USA. Association for Computing Machinery.

[136] Semiconductor, F. MPC8548E PowerQUICC III integrated processor hardware specifications. `http://cache.freescale.com/files/32bit/doc/data_sheet/MPC8548EEC.pdf`.

[137] Shevgoor, M., Koladiya, S., Balasubramonian, R., Wilkerson, C., Pugsley, S. H., and Chishti, Z. (2015). Efficiently prefetching complex address patterns. In *2015 48th Annual IEEE/ACM International Symposium on Microarchitecture (MICRO)*, pages 141–152.

[138] SoCLib (2003-2012). SoCLib: open platform for virtual prototyping of multi-processors system on chip. http://www.soclib.fr/trac/dev.

[139] Soliman, M. R. and Pellizzoni, R. (2017). WCET-Driven Dynamic Data Scratchpad Management With Compiler-Directed Prefetching. In Bertogna, M., editor, *29th Euromicro Conference on Real-Time Systems (ECRTS 2017)*, volume 76 of *Leibniz International Proceedings in Informatics (LIPIcs)*, pages 24:1–24:23, Dagstuhl, Germany. Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik.

[140] Sondag, T. and Rajan, H. (2010). A more precise abstract domain for multi-level caches for tighter wcet analysis. In *2010 31st IEEE Real-Time Systems Symposium*, pages 395–404.

[141] Srinath, S., Mutlu, O., Kim, H., and Patt, Y. N. (2007). Feedback directed prefetching: Improving the performance and bandwidth-efficiency of hardware prefetchers. In *2007 IEEE 13th International Symposium on High Performance Computer Architecture*, pages 63–74.

[142] STMicroelectronics. STM32F756xx datasheet. `http://www.st.com/content/ccc/resource/technical/document/datasheet/fb/d4/56/db/60/61/4f/9c/DM00166114.pdf/files/DM00166114.pdf/jcr:content/translations/en.DM00166114.pdf`.

[143] Strukov, D. (2006). The area and latency tradeoffs of binary bit-parallel bch decoders for prospective nanoelectronic memories. In *2006 Fortieth Asilomar Conference on Signals, Systems and Computers*, pages 1183–1187.

[144] Suhendra, V., Mitra, T., Roychoudhury, A., and Chen, T. (2005). WCET centric data allocation to scratchpad memory. In *26th IEEE International Real-Time Systems Symposium (RTSS'05)*, pages 10 pp.–232.

[145] T., C., P., L., and C., S. K. (2006). Implementation of a pseudo-LRU algorithm in a partitioned cache. Patent US7069390B2.

[146] Technologies, E. (2006). *Efficient Developement of Safe Avionics Software with DO-178B Objectives Using SCADE Suite - Methodological Handbook*. Esterel Technologies.

[147] Tendler, J. M., Dodson, J. S., Fields, J. S., Le, H., and Sinharoy, B. (2002). POWER4 system microarchitecture. *IBM Journal of Research and Development*, 46(1):5–25.

[148] Texas Instruments. TMS570LS09x/07x 16/32-Bit RISC flash microcontroller. `http://www.ti.com/lit/ug/spnu607/spnu607.pdf`.

[149] The Economist. The future of computing. `https://www.economist.com/leaders/2016/03/12/the-future-of-computing`.

[150] Topham, N. and Gonzalez, A. (1999). Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers*, 48(2):185–192.

[151] Trilla, D., Hernandez, C., Abella, J., and Cazorla, F. J. (2016). Resilient random modulo cache memories for probabilistically-analyzable real-time systems. In *2016 IEEE 22nd International Symposium on On-Line Testing and Robust System Design (IOLTS)*, pages 27–32.

[152] Ungerer, T., Bradatsch, C., Frieb, M., Kluge, F., Mische, J., Stegmeier, A., Jahr, R., Gerdes, M., Zaykov, P. G., Matusova, L., Li, Z. J. J., Petrov, Z., Böddeker, B., Kehr, S., Regler, H., Hugl, A., Rochange, C., Ozaktas, H., Cassé, H., Bonenfant, A., Sainrat, P., Lay, N., George, D., Broster, I., Quiñones, E., Panic, M., Abella, J., Hernández, C., Cazorla, F. J., Uhrig, S., Rohde, M., and Pyka, A. (2016). Parallelizing industrial hard real-time applications for the parmerasa multicore. *ACM Trans. Embed. Comput. Syst.*, 15(3):53:1–53:27.

[153] Wartel, F., Kosmidis, L., Gogonel, A., Baldovino, A., Stephenson, Z., Triquet, B., Quiñones, E., Lo, C., Mezzetta, E., Broster, I., Abella, J., Cucu-Grosjean, L., Vardanega, T., and Cazorla, F. J. (2015). Timing analysis of an avionics case study on complex hardware/software platforms. In *2015 Design, Automation Test in Europe Conference Exhibition (DATE)*, pages 397–402.

[154] Wartel, F., Kosmidis, L., Lo, C., Triquet, B., Quiñones, E., Abella, J., Gogonel, A., Baldovin, A., Mezzetti, E., Cucu, L., Vardanega, T., and Cazorla, F. J. (2013). Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *2013 8th IEEE International Symposium on Industrial Embedded Systems (SIES)*, pages 241–248.

[155] Wenzel, I., Kirner, R., Rieder, B., and Puschner, P. (2005). Measurement-based worst-case execution time analysis. In *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, pages 7–10.

[156] Wilhelm, R. (2018). Mixed feelings about mixed criticality (invited paper). In Brandner, F., editor, *18th International Workshop on Worst-Case Execution Time Analysis, WCET 2018, July 3, 2018, Barcelona, Spain*, volume 63 of *OASICS*, pages 1:1–1:9. Schloss Dagstuhl - Leibniz-Zentrum für Informatik.

[157] 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., and Stenström, P. (2008). The worst-case execution-time problem—overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3).

[158] Wilhelm, R., Grund, D., Reineke, J., Schlickling, M., Pister, M., and Ferdinand, C. (2009). Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 28(7):966–978.

[159] Wilkerson, C., Alameldeen, A. R., Chishti, Z., Wu, W., Somasekhar, D., and Lu, S. (2010). Reducing cache power with low-cost, multi-bit error-correcting codes. In *Proceedings of the 37th Annual International Symposium on Computer Architecture*, ISCA '10, page 83–93, New York, NY, USA. Association for Computing Machinery.

[160] Wilkerson, C., Gao, H., Alameldeen, A. R., Chishti, Z., Khellah, M., and Lu, S. L. (2008). Trading off cache capacity for reliability to enable low voltage operation. In *2008 International Symposium on Computer Architecture*, pages 203–214.

[161] Wilson, A. and Preyssler, T. (2008). Incremental certification and integrated modular avionics. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, pages 1.E.3–1–1.E.3–8.

[162] X., V., J., A., A., G., and R., R. (2007). Reducing soft error vulnerability of data caches. In *3rd Workshop on Silicon Errors in Logic - System Effects (SELSE).*

[163] Xilinx. Xilinx zynq ultrascale+ mpsoc data sheet. `https://www.xilinx.com/support/documentation/data_sheets/ds891-zynq-ultrascale-plus-overview.pdf`.