Enabling Caches in Probabilistic Timing Analysis

Leonidas Kosmidis

Computer Architecture Department

Universitat Politècnica de Catalunya

A thesis submitted for the degree of

PhD in Computer Architecture

30th of May, 2017

Enabling Caches in Probabilistic Timing Analysis

Leonidas Kosmidis
May 2017

Universitat Politècnica de Catalunya
Computer Architecture Department

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

Advisors: Eduardo Quiñones, Barcelona Supercomputing Center (BSC), (Advisor)
Jaume Abella, Barcelona Supercomputing Center, (Co-advisor)
Francisco Cazorla, IIIA-CSIC and BSC, (Co-advisor)
Mateo Valero, Universitat Politècnica de Catalunya and BSC, (Tutor)

The work reported in this thesis has been conducted at the Computer Architecture and Operating Systems Interface (CAOS) group of the Barcelona Supercomputing Center (BSC), and has been financially supported by the Spanish Ministry for Education under the FPU grant AP-2010-4208 and by the European Commission through the FP7 projects PROARTIS (FP7-ICT1.3.4-249100) and PROXIMA (611085).

To my tireless companion
in the beautiful journey of life,
my wife, Maritina

Acknowledgements

Σὰ βγεῖς στὸν πηγαιμὸ γιὰ τὴν Ἰθάκη,
νὰ εὔχεσαι νᾅναι μακρὺς ὁ δρόμος,
γεμάτος περιπέτειες, γεμάτος γνώσεις.
[...]
Κι ἂν πτωχικὴ τὴν βρῇς, ἡ Ἰθάκη δὲν σὲ γέλασε.
Ἔτσι σοφὸς ποὺ ἔγινες, μὲ τόση πείρα,
ἤδη θὰ τὸ κατάλαβες ἡ Ἰθάκες τί σημαίνουν.

Ιθάκη, Κωνσταντίνος Π. Καβάφης (1911)

As you set out for Ithaka
hope the voyage is a long one,
full of adventure, full of discovery.
[...]
And if you find her poor, Ithaka won't have fooled you.
Wise as you will have become, so full of experience,
you will have understood by then what these Ithakas mean.

Ithaka, Constantine P. Cavafy (1911)
(Translation by Edmund Keeley/Philip Sherrard)

Six years, two European and an ESA project later a journey comes to its end. A journey full of learning, hard work and innovation.

The first years were rough. A couple of paper rejections, the difficulty to convince the critical real time systems community for a solution so different from the traditional way followed for decades. Eventually came redemption, to replace the initial disappointment with joy and relief. I could never imagine back then, that there will be one day that the proposals of this thesis would be widely accepted and even establish a new field in the real-time systems research and implementation.

This thesis wouldn't be possible without the PROARTIS project, which brought together exceptional researchers from different backgrounds to collaborate intensively and invest significant resources to a groundbreaking and controversial idea. This collaborative effort laid the foundations of Probabilistic Timing Analysis and in particular the Measurement-Based variant (MBPTA) which this thesis is based on and offered me the opportunity to propose hardware and software solutions to make it possible in practice. I'm proud to have been part of the initial PROARTIS team and its industrialisation-focused successor PROXIMA, and have contributed significantly to their success.

Of course, this would have never happened without my advisors: Dr. Eduardo Quiñones, Dr. Jaume Abella, Dr. Francisco Cazorla and prof. Mateo Valero. First of all, I would like to thank them for making the aforementioned projects possible with successful project and grant proposals, and with the effective coordination of their activities. Second, I would like to express my gratitude for having given me the opportunity to participate in those innovative European and ESA funded projects – PROARTIS, PROXIMA and PROARTIS for SPACE (P4S) – and for having proposed me this interesting and challenging topic. I am also grateful for their constant assistance during my research in any form; from the endless discussions and explanations to the encouragement they gave me until the last moment. Next to them I learned not only how to do research but also what I should avoid in this endeavour.

During these years I had the opportunity to meet and collaborate with exceptional people: prof. Tullio Vardanega from University of Padua has been an extraordinary teacher from whom I learned a lot on real-time systems theory and implementation as well as project management; his PhD students (led by my friend Dr. Enrico Mezzetti: Andrea Baldovin, Davide Compagnin, Luca Bonato, Marco Ziccardi) provided the RTOSes for several case studies running on the platforms which were hardware or software randomised based on the contributions of this thesis; Airbus France folks Franck Wartel and Benoît Triquet who always made me to feel welcome in every visit at Airbus facilities in Toulouse. They taught me a lot about the PowerPC architecture internals and avionics systems, and they scrutinised every bit of my proposals to ensure their applicability to the avionics domain; all INRIA group members led by Dr. Liliana Cucu with whom we established the first version of MBPTA theory upon which this thesis is built; prof. Emery Berger and his student Charlie Curtsinger who implemented Stabilizer, the tool we used to implement dynamic software randomisation.

All the people of the CAOS group with whom we shared an office and we had a wonderful time all these years. They are not only good colleagues but also good friends. From those I need to specially thank for their direct or indirect technical contributions on this thesis: Mikel Fernandez who got the responsibility to maintain and extend the simulator I built for my thesis and PROARTIS, so that it could be widely used in our group as well as in other European and ESA funded projects; David Morales who under my supervision continued my work on dynamic software randomisation by porting it to the SPARC LEON3 FPGA board and by integrating it with various industrial RTOSes and helped to collect results with TASA; Roberto Vargas who prototyped the TASA parser in a couple of days and with whom we spent endless hours exchanging opinions and solutions on technical matters; Dr. Carles Hernandez who implemented the hardware proposals of this thesis in the LEON 3 RTL together with Cobham Gaisler colleagues to produce the first ever MBPTA-compliant processor in the market.

Since the process of obtaining a PhD title is the uttermost celebration of education, in addition to the people related to my doctoral studies, I would like to thank all the people that affected my education since my young age to shape my character and to continue pursuing knowledge during my entire life. Similar to Alexander the Great, one of the most important figures of both Greek and global history, who used to praise his tutor Aristotle, saying that he owed his existence to his father and his welfare to his tutor. [Plutarch, Parallel Lives, Life of Alexander, Section 8.4]. Starting from my uncle Eftychios who when I was 6 years old he gifted me my first encyclopedia and he wrote in his dedication: "A very small stone to the building of your knowledge". Thank you uncle for the solid foundations! I would like also to thank my family members, friends and my teachers across all educational levels that helped me to become the man I am, either by supporting me, or teaching me principles, life lessons or knowledge of any kind.

Raised in the Greek public educational system – the gem of our small country, which has produced and continues to produce many brilliant scientists and it must be protected and improved as an investment for the future generations – from primary school to my bachelor's degree, I was blessed to have received a very high quality education. I had exceptional educators in their entirety, full of passion about their job and love for their students, who were trying their best every day even with the limited resources they were provided with.

From those I would like to specifically mention my advisors at my alma mater, the Computer Science Department of University of Crete, prof. Angelos Bilas and prof. Manolis Katevenis, for giving me solid foundations on computer architecture and creating the spark for pursuing a doctoral degree in this fascinating area.

Last but not least, I want to thank my beloved wife Maritina, for her patience and her continuous support and encouragement during the period of my doctoral thesis' research. She offered me a warm family and she makes my life beautiful every single day.

This work has been funded by the Spanish Ministry for Education under the FPU grant AP-2010-4208, the European Commission through the projects PROARTIS (FP7-ICT1.3.4-249100) and PROXIMA (611085).

Abstract

Hardware and software complexity of future critical real-time systems challenges the scalability of traditional timing analysis methods. Measurement-Based Probabilistic Timing Analysis (MBPTA) has recently emerged as an industrially-viable alternative technique to deal with complex hardware/software. Yet, MBPTA requires certain timing properties in the system under analysis that are not satisfied in conventional systems. In this thesis, we introduce, for the first time, hardware and software solutions to satisfy those requirements as well as to improve MBPTA applicability. We focus on one of the hardware resources with highest impact on both average performance and Worst-Case Execution Time (WCET) in current real-time platforms, the cache. In this line, the contributions of this thesis follow three different axes: hardware solutions and software solutions to enable MBPTA, and MBPTA analysis enhancements in systems featuring caches.

At hardware level, we set the foundations of MBPTA-compliant processor designs, and define efficient time-randomised cache designs for single- and multi-level hierarchies of arbitrary complexity, including unified caches, which can be time-analysed for the first time.

We propose three new software randomisation approaches (one dynamic and two static variants) to control, in an MBPTA-compliant manner, the cache jitter in Commercial off-the-shelf (COTS) processors in real-time systems. To that end, all variants randomly vary the location of programs' code and data in memory across runs, to achieve probabilistic timing properties similar to those achieved with customised hardware cache designs.

We propose a novel method to estimate the WCET of a program using MBPTA, without requiring the end-user to identify worst-case paths and inputs, improving its applicability in industry. We also introduce Probabilistic Timing Composability, which allows Integrated Systems to reduce their WCET in the presence of time-randomised caches.

With the above contributions, this thesis pushes the limits in the use of complex real-time embedded processor designs equipped with caches and paves the way towards the industrialisation of MBPTA technology.

Contents

List of Figures

Chapter 1

Introduction

For decades computers have been used almost exclusively for large scale scientific computation, business/office automation and entertainment. Recently, this trend has abruptly changed with computers intervening in a growing number of critical aspects of human life related to health, finance, security and safety among others [Duranton *et al.* (2013)][Duranton *et al.* (2015)][Girbal *et al.* (2013)]. The digitisation of activities critical to the society brings huge potential benefits including personalised healthcare, safer and lower-emission transportation, and improvements in cost reduction in industrial production [ARTEMIS, ITEA, and EUREKA (2015)]. For this reason, the embedded computing systems domain, which includes handhelds, Internet of Things (IoT) devices and control systems, is experiencing an unprecedented growth. In particular, while in the past the larger share of the semiconductor industry revenue came from the high-performance and server market, nowadays the embedded computing is the main contributor, forcing big industry players to reconsider their strategy [Forbes (2015)] [Financial Times (2015)].

However, to effectively consolidate the new digitisation opportunities, changes are required at both hardware and software level of embedded systems and more specifically to their Critical Real-Time Embedded Systems (CRTES) subset. These changes go in the line of increasing time analysability: unlike other computing domains, CRTES do not require only functional correctness, but time plays an important role for correct operation too, since tasks performed by CRTES have specific time boundaries, called *deadlines*. The potential impact of increased analysability is significant since CRTES are present in markets such as avionics, automotive, railway, aerospace, telecommunication and medical, and constitute a large share of the embedded market. Further, the CRTES domain has experienced an unprecedented growth in the last decade and this trend is expected to continue in the future. In the automotive domain alone, it is estimated that the annual production will reach the 100,000,000 vehicles per year, by 2020, increasing the global number of existing automobiles over 50% [Jeffrey Owens, Delphi Automotive (2015)].

Last decades have also witnessed a dramatic change in the way CRTES are designed, transitioning from federated architectures to integrated architectures. Taking as example the avionics domain, in the past, conventional avionics systems were based on the *federated architecture* paradigm [Hoyme & Driscoll (1992)], in which each computer was a fully dedicated unit to execute a single system function. However, due to the need to reduce costs – in terms of size, weight and power as well as development and maintenance costs – and since most of the federated computers perform essentially the same functions (input acquisition, processing and output generation), a natural optimisation of resources is to share the development effort by identifying common subsystems, standardising interfaces and encapsulating services; in other words, adopting a modular approach. That is the intent of the Integrated Modular Avionics (IMA) concept [Prisaznuk (1992)] [Pelton & Scarbrough (1997)], whose integration refers to the sharing of (platform) resources by multiple subsystems. Therefore, federated architectures have been replaced by *Integrated Architectures* [Watkins & Walter (2007)], in which the same computer can host multiple applications, potentially operating at distinct criticality levels (*mixed-criticality*). Although this paradigm shift comes from the IMA concept inbred to the avionics community, it is also applicable to other application domains. In the automotive sector for example, software components can be supplied from multiple sources, integrated on the same hardware platform or physically distributed and possibly moved from one CPU to another without loss of functional and time correctness, while also providing a guaranteed level of reliability [Di Natale & Sangiovanni-Vincentelli (2010)].

1.1 Specific Requirements

Integrated mixed-criticality CRTES have specific requirements including – but not limited to – high performance, and timing analysability and composability. Below we examine those requirements and their implications on the CRTES design.

1.1.1 High Performance

Although CRTES were traditionally mechanical, they have gradually incorporated more and more electronic controls in critical operations, like engine and brake management. Further, every new generation features new functionalities, e.g. driver assistance systems in the automotive industry such as Anti-lock Braking Systems (ABS), traction control, and collision avoidance up to fully autonomous self-driving systems. Hence, critical software is increasingly used for autonomous operation and decision-making in all domains. As a result critical software is increasing in complexity and computing performance requirements. For example in the avion-

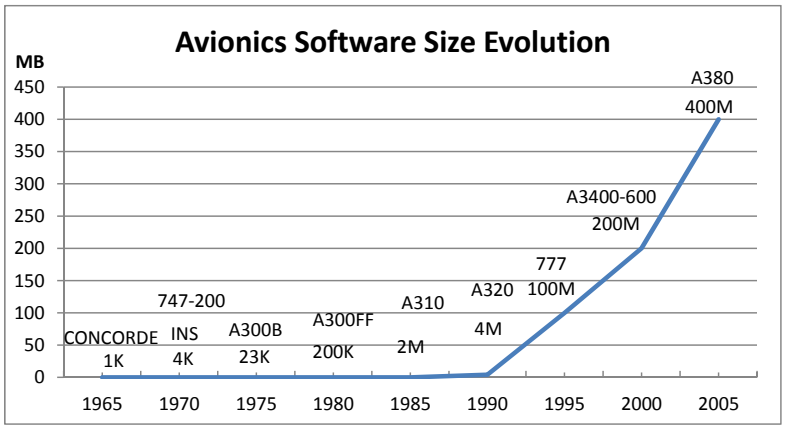


Figure 1.1: Evolution of avionics software size.

ics domain (Figure 1.1), the size of aircraft software has been doubling in size approximately every two years up to 1990, while after that it has been still growing with an exponential rate [Potocki de Montalk (1991)] [Butz (2007)]. Current trends show that critical software will consume highest ever levels of computing performance, e.g. advanced driver assistance systems are predicted to use at least 100x more compute performance by 2024 compared to 2016 systems [Winstanley (2015)].

In order to satisfy the increasing computational power requirements of those new functionalities (while trying to reduce power, size and cost and keep competitiveness in the market) CRTES industries started increasing the number of processing elements in their products. However, this resulted in a rapid multiplication of the computational elements used: for example, as of 2014, a high-end car includes more than 100 ECUs (Electronic Control Units) [Jim Tung, MathWorks, Inc and James Buczkowski, Ford Motor Company (2014)].

The processors used in CRTES of these systems were – and many of them are still today – microcontrollers with very simple architectures [Charette (2009)] [Fleming (2011)] [Wang *et al.* (2012)], lacking many performance enhancing features found in modern processors (e.g. memory hierarchies or branch predictors). As a result, the only way to cover the increased performance demands of critical software has been by replication. This leads designers not only to face a scalability wall, but also increased reliability concerns, since the multiplication of ECUs requires a higher number of less-reliable elements such as cables and connectors.

In this scenario, the only viable solution for the CRTES industry to attain these unprecedented performance needs at competitive costs, is by using aggressive computer designs including parallel complex multicore platforms. The other side of the coin is that the massive use of stateful resource features in high-performance hardware heavily complicates to provide guaranteed performance, which is already

an extremely time-consuming step in the typical embedded design process using much simpler processor designs, as discussed in the following section.

1.1.2 Timing Analysability

CRTES follow a strict validation and verification process, which is directly related to the organisation of those systems [Littlewood & Strigini (1993)] [Knutson & Carmichael (2001)] [Michael *et al.* (2011)]. These necessary verification and validation steps of CRTES are performed before system deployment and they are estimated to consume almost 50% of the development cost [Croxford & Sutton (1996)]. Furthermore, the use of complex hardware can easily make this percentage to reach unaffordable values.

Complex hardware impacts negatively the existing static and measurement-based (deterministic) timing analysis techniques: Static deterministic analysis methods (SDTA), further detailed in the Background Chapter, are challenged by the increasing complexity of modern systems' internal state: while each hardware component may have deterministic behaviour, their complex relation is hard to track and model. Moreover, complex hardware cause an explosion in the number of possible hardware states, which makes SDTA to face scalability issues [Mezzetti & Vardanega (2011b)] [Nowotsch *et al.* (2014)]. In addition, the lack of details on processor internals, due to intellectual property restrictions or incomplete specifications, limits the information available for analytical timing models [Abella *et al.* (2015)]. Those models, therefore, resort to worst-case assumptions to account for the unknown, leading to pessimistic predictions. To favour SDTA, simple hardware is proposed to be used in the design of real-time systems [Thiele & Wilhelm (2004)]. However, these proposals include severe design changes and performance-capping features that result in average performance loss and, therefore, are not adopted in general by the mainstream processor design industry.

Measurement-based deterministic timing analysis (MBDTA) techniques are based on the execution time observation of the program on the actual hardware, in order to measure the high-watermark execution time. An engineering margin based on user experience is added on this longest execution time to obtain a WCET bound to cover unknown parameters that can potentially increase the execution time [Wartel *et al.* (2013)]. However, complex hardware complicates the selection of an appropriate reliable margin. This is caused by the difficulty to derive evidence that worst-case system behaviour is captured in the measurement runs, which impacts negatively the users' confidence on the derived timing bounds. Although there exist MBDTA variants which take into account the structure of the program to increase the confidence over the WCET estimate [Rapita Systems (2008)], they face the same limitations when applied on high-performance complex hardware.

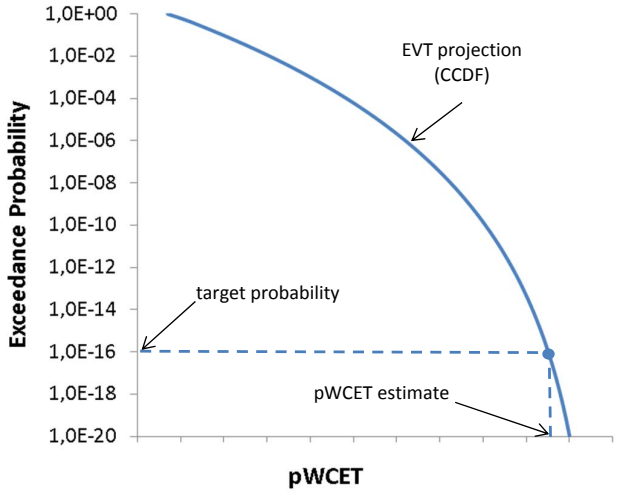


Figure 1.2: Example of pWCET distribution.

Probabilistic Timing Analysis (PTA) [Cazorla *et al.* (2013a)] [Hansen *et al.* (2009)] [Cucu-Grosjean *et al.* (2012)] has recently emerged as an attractive alternative to existing techniques to handle the complexity of high-performance hardware. The key differentiating element with traditional Deterministic-Timing Analysis (DTA) techniques is that, rather than computing a single WCET value, PTA derives a probabilistic WCET (pWCET) distribution that assigns to each WCET estimate the probability of the program to overrun its deadline. An example of such distribution is shown in Figure 1.2 in the form of a Complementary Cumulative Distribution Function (CCDF). Note that the probability distribution can reach arbitrary low values and the target probability can be selected as low as required, based on the specification of the analysed system.

PTA considers timing bounds in the same manner as the embedded safety-critical systems domain addresses system reliability, which is expressed as a compound function of the probabilities of hardware failures and software faults. PTA extends this probabilistic notion to timing correctness. PTA seeks WCET bounds for arbitrarily low probabilities, so that even if violation events may in principle occur, they would only do with a probability well below system safety requirements as defined in safety standards for CRTES [International Organization for Standardization (2009)] [RTCA and EUROCAE (1992)]. Indeed, a probability of exceedance in the region of 10^{-50} per activation for a critical software function is well below the probability of a meteorite falling on the aircraft.

However, Probabilistic Timing Analysis and, in particular, its Measurement-Based variant (MBPTA) [Cucu-Grosjean *et al.* (2012)], which we consider in this Thesis, cannot be naively used on every system. It requires certain timing properties not provided in current conventional hardware/software platforms. First, all

the jittery hardware elements (i.e. those with variable latency) have to be controlled so that their impact on execution time is captured in the tests performed at analysis. The end goal is that, the jitter captured in the measurements must upperbound the one that can potentially occur during system operation. Second, the execution times of programs during analysis – which are made to upperbound those during operation by construction – need to have a distinct probability of occurrence so that their behaviour can be modelled by a random variable. This random variable needs to be *independent and identically distributed* (i.i.d), which means that the observed execution times are independent among them and must follow the same probability distribution. This second requirement originates from the fact that the execution times, which are collected in the same manner as in conventional Measurement-Based timing analysis, are subsequently processed by well established mathematical tools such as Extreme Value Theory (EVT) [Kotz & Nadarajah (2000)] in order to upper-bound the extremes of the execution time distribution.

1.1.3 Time Composability

One fundamental requirement of integrated architectures (e.g. IMA in avionics) is to enable *incremental qualification* [RTCA (2005)] [Wilson & Preyssler (2008)] [Elmqvist *et al.* (2008)], whereby each partition (functional subsystem) can be subject to verification and validation – including timing analysis – in isolation, independent of the other partitions, with obvious benefits for cost, time and effort. From the perspective of timing analysis, incremental qualification rests on the hypothesis that each hardware and software component that is part of the system exhibits the property of *time composability* at run time [Puschner & Schoeberl (2008)] [Puschner *et al.* (2009)].

In the most general definition, composability ensures that, given a property of each item of a collection, that property can be determined for each item taken in isolation and it does not change when that item is brought together with other items. Time composability refers to the fact that the execution time of a software partition, determined in isolation by the timing analysis, is not affected by the presence of other partitions in the same system. However, the access to physical execution resources may introduce run time dependence across partitions, which breaks time composability. That is, the execution time of a partition that accesses a given hardware resource (e.g., the cache) may depend on the state of that resource as left by previous accesses from other partitions. As a result, execution time may vary depending on the actual scheduling of software components, which in IMA is a function of system integration. IMA platforms achieve single-core temporal isolation by flushing the cache at partition switch and assuming pessimistic worst-case bounds for the response time of Operating System (OS) services.

1.2 Benefits and Challenges of Caches

Caches are one of the most, if not the most, important resource when it comes to improve the performance of a processor architecture. However, cache memories represent one of the biggest challenges in timing analysis, because their timing heavily depends on their internal state and it introduces large variations in the execution time. In fact, cache memories have been shown to significantly impact average and WCET, and have been historically acknowledged as one of the most important elements impacting WCET estimation [Mueller & Harmon (1993)] [Ferdinand & Wilhelm (1999)] [Mueller (2000)] [Ferdinand *et al.* (2001)] [Lesage *et al.* (2009)] [Hardy & Puaut (2008)]. Moreover, cache resources are abundant in a processor architecture, including the first level instruction and data caches, translation lookahead buffers (TLBs), second level caches, etc.

A memory access in the presence of a cache has different latencies depending on whether the corresponding data are present in the cache (hit) or not (miss). These two latencies usually differ in orders of magnitude. Therefore, pessimistic assumptions regarding accesses, can degrade Worst-Case Execution Time (WCET) estimates very fast.

SDTA for caches is based on abstract interpretation [Cousot & Cousot (2004)] in order to perform *cache analysis* [Ferdinand & Wilhelm (1999)]. This analysis consists of three separate sub-analyses: must analysis, may analysis and persistence analysis. These analyses perform a classification of each memory access. The first determines whether a cache block is always present in the cache (hit), the second whether it may be in the cache and the latter whether it is not evicted after it is loaded. It has been shown that this process does not scale with large industrial size programs since it requires huge amount of data for this processing. More importantly though, it relies on knowing the exact memory addresses, which is not always possible. As a consequence, lack of information impacts significantly the tightness of the WCET estimates [Abella *et al.* (2014a)]. For example, even a single access to an unknown memory address with a 4-way set associative cache, leads static approaches to assume that an entire way of each set has been invalidated, because it is unable to determine which set was accessed and provably evicted. In other words, one quarter of the effective cache size is assumed lost.

MBDTA methods face difficulties in providing confidence on minimum guaranteed hit rates based only on observations. The main reason for this is that, due to the inherent properties of conventional cache designs, which exploit temporal and spatial locality, usually the observed hit rates are high. However, under special circumstances, particular memory access patterns can yield very low hit rates (cache thrashing), known as *pathological cases* or *cache risk patterns* [Vardanega *et al.* (2007)][E.Mezzetti *et al.* (2008)], resulting in unusually long execution times. Since these events happen very rarely and can only be occasionally observed, they

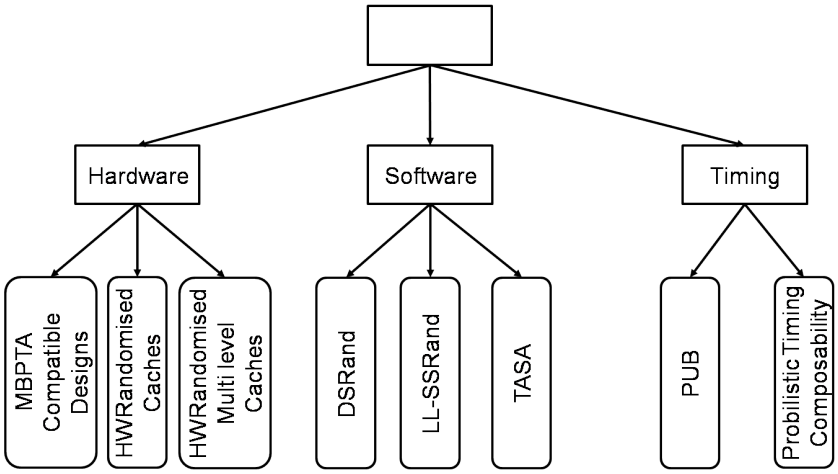


Figure 1.3: Logical Organisation of this Thesis' contributions.

cannot be quantified in existing cache designs [Quiñones *et al.* (2009)], therefore reducing the confidence of the obtained values. In fact, in order to compensate for these cases, as well as for other system unknowns, the current industrial practice is to add an engineering margin (e.g. 20% in some avionics systems [Wartel *et al.* (2013)]) over the highest observed execution times, resulting in system over-provisioning and still unknown confidence.

1.3 Contributions

This Thesis sets the foundations of MBPTA compatible designs, by demonstrating how processor components used in a processor architecture can be modified in order to comply with its requirements. Subsequently, this Thesis applies these modifications on a particular resource, the cache. Further this Thesis focuses on proposing hardware/software designs that enable the use of caches in the context of Probabilistic Timing Analysis techniques. Moreover, this Thesis elaborates on the implications of the cache as a central element in those designs. The ultimate goal in this process is to satisfy both, MBPTA requirements as well as modern critical systems design principles such as integrated systems, in order to facilitate its adoption by the CRTES industry. The contributions of this Thesis can be divided into three major themes, as they are visually depicted in Figure 1.3: Hardware Solutions for MBPTA compatibility; Software Solutions for enabling the use of existing systems with caches in MBPTA; and Timing Analysis Aspects.

1.3.1 Hardware Solutions

The first contribution of the Thesis is to set the foundations of MBPTA compatible processor designs. We define a taxonomy of hardware resources based on their timing behaviour, categorising them according to their possible working latencies (jitter). In the case of buffers and other jitter-less resources, which only propagate latencies, we show that they do not break MBPTA requirements [Kosmidis *et al.* (2013e)]. Next, we show that jittery resources (i.e. with different possible latencies) can be either time-randomised, when these latencies differ significantly as it is the case for caches, or, when their jitter is small, e.g. floating point-units, redesigning them to work on their worst latency [Kosmidis *et al.* (2014d)].

Secondly, we propose hardware solutions with novel and efficient cache organisations that are compatible with Probabilistic Timing Analysis. We propose a low-overhead random placement policy [Kosmidis *et al.* (2013a)] [Kosmidis *et al.* (2014a)] that enables the use of Measurement-Based Probabilistic Timing Analysis in single level direct-mapped and set-associative cache structures, for use either for instruction or data caches and TLBs.

Next we extend this concept to arbitrarily complex memory hierarchies [Kosmidis *et al.* (2013b)], by studying various cache configurations that can be found in systems with multiple levels of cache. In particular, we study inclusive and non-inclusive caches, as well as write-back and write-through policies. We show that PTA can be used to effectively analyse these complex organisations. Furthermore, we also show, for the first time, that the timing analysis of unified instruction and data caches is possible.

1.3.2 Software Solutions

Although attractive, hardware solutions cannot be applied to legacy CRTES and naturally take longer to hit the market than software solutions. To cover this gap we propose a software-only compiler-based solution [Kosmidis *et al.* (2013c)], which randomises the placement of memory objects (instructions and stack) in memory dynamically at runtime. For this reason we term it Dynamic Software Randomisation (DSRand in Figure 1.3). This way, the mapping of the objects in cache becomes random, and so do cache evictions among them. We show that using this method, we enable the use of MBPTA on conventional caches.

The dynamic nature of this technique, though, challenges the certification of this technique in automotive systems, since self-modifying software creates some difficulties in its adoption in such systems. For this reason, in [Kosmidis *et al.* (2014c)], we propose a modification of this method, applied in an entirely static manner and named Link-Level Static Software Randomisation (LL-SSRand in Figure 1.3). This is achieved by a combination of compiler transformations together with link time randomisation of functions placement in the binary.

Despite that the previous method is more amenable to certification than the dynamic variant, it still consists of a compiler part which requires the requalification of existing compiler toolchains. Moreover, changing an industrial compiler toolchain to enable MBPTA is not always possible, due to their closed source nature, and in addition it needs to be retargeted for each supported platform. In order to overcome this limitation, we propose TASA (Toolchain Agnostic Software Randomisation) [Kosmidis *et al.* (2016c)], which applies static software randomisation at source-code level, being independent of the underlying compiling infrastructure.

1.3.3 Timing Analysis aspects related to the use of caches in MBPTA

The use of MBPTA on real systems, which has been achieved with the hardware and software proposals of this Thesis, opens the door for cost effective optimisations on large scale industrial systems, like the ones used in avionics.

Conventional MBPTA results [Cucu-Grosjean *et al.* (2012)] are valid only for the exercised paths at analysis time. To relieve users from the need to provide full path coverage or to identify inputs leading to WCET, which is also the case in traditional deterministic timing analysis (DTA), we propose a software method called Path Upper-bounding (PUB) [Kosmidis *et al.* (2014b)], which probabilistically upper-bounds the execution time of any path of the program independently from the input.

Moreover, due to the complexity of those industrial software architectures, the development and the timing verification of individual software parts are performed independently. In order to guarantee that the timing analysis results are valid in the presence of *composition*, regardless of the hardware state left in the processor from the previously executed piece of software, the hardware state, and especially cache contents, is assumed to be in its worst state (e.g., empty caches). However, the particular structure of avionics systems requires the repetitive execution of software units, which leaves a lot of potential for reusing cache contents across executions, something not possible with traditional timing analysis methods. The use of MBPTA gives the opportunity to exploit these contents, hence reducing the WCET of units of composition, and therefore increasing the schedulability of the system. In order to achieve this, we propose a method to obtain *Probabilistic Timing Composability* [Kosmidis *et al.* (2013d)].

1.4 Structure of the Thesis

We keep one contribution per chapter and present them in the aforementioned order. In addition, since several of the Thesis contributions have been recently validated in industrial setups, each relevant chapter contains a short summary of the obtained results and refers to the appropriate published articles.

- In Chapter 2 we present the necessary Background on Real-Time Systems and a survey of the various Timing Analysis methods which are used to estimate the WCET. Special attention is given to Probabilistic Methods and, especially, MBPTA and its requirements upon which this Thesis is built.
- In Chapter 3 we describe the experimental setup used for the evaluation of the contributions of this Thesis, as well as the followed methodology.
- In Chapter 4 we introduce the problem statement – the requirements that MBPTA places on processor architectures–, we present the taxonomy of processor resources and we show their required changes to achieve MBPTA compliance.
- Chapters 5 and 6 are dedicated to hardware proposals for cache designs, with the former focusing on single-level cache designs and the latter on multi-level cache hierarchies.
- Chapters 7 to 9 describe software solutions, starting with Dynamic Software Randomisation, followed by static variants, at link-level and source-level (TASA). Chapter 9 includes also a quantitative comparison between Dynamic Software Randomisation and TASA.
- Chapters 10 and 11 focus on the timing analysis aspects related to the uses of caches in MBPTA. The former introduces the Path Upper-bounding technique (PUB) while the latter presents Probabilistic Timing Composability.
- Finally Chapter 12 draws the conclusions of the Thesis, quantifies its impact and concludes with future directions for research in this area.

1.5 List of Publications

Below we list the publications that the research of this Thesis has produced.

1.5.1 Accepted Publications

- **L. Kosmidis**, R. Vargas, D. Morales, E. Quiñones, J. Abella, F. J. Cazorla. *TASA: Toolchain Agnostic Software Randomisation.* International Conference on Computer Aided Design (ICCAD), Austin, Texas, November 2016.

- **L. Kosmidis**, J. Abella, E. Quiñones, F. Wartel, G. Farrall, F. J. Cazorla. *Containing Timing-Related Certification Cost in Automotive Systems Deploying Complex Hardware.* **Best Paper Award.** In Design Automation Conference (DAC), San Francisco, CA, June 2014.

- **L. Kosmidis** , E. Quiñones, J. Abella, T. Vardanega, I. Broster, F. J. Cazorla. *Measurement-Based Probabilistic Timing Analysis and Its Impact on Processor Architecture.* In Euromicro Conference on Digital System Design (DSD), Verona, Italy, August 2014.

- **L. Kosmidis**, J. Abella, E. Quiñones, Francisco J. Cazorla. *Efficient Cache Designs for Probabilistically Analysable Real-Time Systems.* In IEEE Transactions on Computers (ToC), Volume 63, Issue 12, 2998-3011, 2014.

- **L. Kosmidis**, J. Abella, F. Wartel, E. Quiñones, A. Collin, F. J. Cazorla. *PUB: Path Upper-Bounding for Measurement-Based Probabilistic Timing Analysis.* In Euromicro Conference on Real-Time Systems (ECRTS), Madrid, Spain, July 2014.

- **L. Kosmidis**, J. Abella, E. Quiñones, F. J. Cazorla. *Multi-Level Unified Caches for Probabilistically Time Analysable Real-Time Systems.* In IEEE Real-Time Systems Symposium (RTSS), Vancouver, Canada, December 2013.

- **L. Kosmidis**, T. Vardanega, J. Abella, E. Quiñones and Francisco J. Cazorla. *Measurement-Based Probabilistic Timing Analysis to Buffer Resources..* In 13th International Workshop on Worst-Case Execution Time Analysis (WCET), Paris, France, July 2013.

- **L. Kosmidis**, E. Quiñones, J. Abella, T. Vardanega, and F. J. Cazorla. *Achieving Timing Composability with Probabilistic Timing Analysis.* The 16th IEEE Computer Society Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing (ISORC), June 2013.

- **L. Kosmidis**, J. Abella, E. Quiñones, and F. J. Cazorla. *A Cache Design for Probabilistically Analysable Real-Time Systems.* Design, Automation, and Test in Europe (DATE), Grenoble, France, March 2013.

- **L. Kosmidis**, C. Curtsinger, E. Quiñones, J. Abella, E. Berger, and F. J. Cazorla. *Probabilistic Timing Analysis on Conventional Cache Designs.* Design, Automation, and Test in Europe (DATE), Grenoble, France, March 2013.

1.5.2 Other Publications

Below we list other publications that, although do not constitute contributions of this Thesis, are directly related to it. These works can be classified in three categories:

a) Works validating the solutions proposed and developed in this Thesis in a realistic industrial environment, by using industrial case studies executed on the simulation environment created in this Thesis or real hardware platforms. The real hardware platforms include both Commercial off-the-shelf (COTS) processors, as well as an RTL-implementation of the hardware proposals introduced in this Thesis on an FPGA.

b) Works continuing research on time-randomised hardware and software randomisation building on the foundations laid by this Thesis.

c) Publications related to the fundamental theory of Probabilistic Timing Analysis and especially its Measurement-Based variant which this Thesis is focused on.

- F. Cros, **L. Kosmidis**, F. Wartel, D. Morales, J. Abella, I. Broster, F. J. Cazorla. *Dynamic Software Randomisation: Lessons Learned From an Aerospace Case Study.* In 20th Design Automation and Test in Europe Conference (DATE), Lausanne, Switzerland, March 2017.

- M. Fernandez, D. Morales, **L. Kosmidis**, A. Bardizbanyan, I. Broster, C. Hernandez, E. Quiñones, J. Abella, F. J. Cazorla, P. Machado, L. Fossati. *Probabilistic Timing Analysis on Time-Randomized Platforms for the Space Domain.* In 20th Design Automation and Test in Europe Conference (DATE), Lausanne, Switzerland, March 2017.

- F. J. Cazorla, J. Abella, J. Anderson, T. Vardanega, F. Vatrinet, I. Bate, I. Broster, M. Azkarate-Askasua, F. Wartel, L. Cucu, F. Cross, G. Farrall, A. Gogonel, A. Gianarro, B. Triquet, C. Hernandez, C. Lo, C. Maxim, D. Morales, E. Quiñones, E. Mezzetti, **L. Kosmidis**, I. Agirre, M. Fernandez, M. Slijepcevic, P. Conmy and W. Talaboulma. *PROXIMA: Improving*

Measurement-Based Timing Analysis through Randomisation and Probabilistic Analysis. In Euromicro Conference on Digital System Design (DSD), Limassol, Cyprus, August 2016.

- P. Benedicte, **L. Kosmidis**, E. Quiñones, J. Abella, F. J. Cazorla. *A Confidence Assessment of WCET Estimates for Software Time Randomized Caches.* In International Conference on Industrial Informatics (INDIN), Futuroscope-Poitiers, France, July 2016.

- **L. Kosmidis**, D. Compagnin, D. Morales, E. Mezzetti, E. Quiñones, J. Abella, T. Vardanega and F. J. Cazorla. *Measurement-Based Timing Analysis of the AURIX Caches.* In WCET Analysis Workshop (WCET), Toulouse, France, July 2016.

- P. Benedicte, **L. Kosmidis**, E. Quiñones, J. Abella and F. J. Cazorla. *Modeling the Confidence of Timing Analysis for Time Randomised Caches.* In 11th IEEE International Symposium on Industrial Embedded Systems (SIES), Krakow, Poland, May 2016.

- **L. Kosmidis**, E. Quiñones, J. Abella, T. Vardanega, C. Hernandez, A. Gianarro, I. Broster, F. J. Cazorla. *Fitting Processor Architectures for Measurement-Based Probabilistic Timing Analysis.* In ELSEVIER Microprocessors and Microsystems - Embedded Hardware Design. Vol 47, Part B. 2016.

- F. Wartel, **L. Kosmidis**, A. Gogonel, A. Baldovin, Z. Stephenson, B. Triquet, E. Quiñones, C. Lo, E. Mezzetti, I. Broster, J. Abella, L. Cucu-Grosjean, T. Vardanega, F. J. Cazorla. *Timing Analysis of an Avionics Case Study on Complex Hardware/Software Platforms.* In 18th Design, Automation and Test in Europe Conference (DATE), Grenoble (France), March 2015.

- M. Slijepcevic, **L. Kosmidis**, J. Abella, E. Quiñones, F. J. Cazorla. *Time-Analysable Non-Partitioned Shared Caches for Real-Time Multicore Systems.* In Design Automation Conference (DAC), San Francisco, CA, June 2014.

- J. Jalle, **L. Kosmidis**, J. Abella, E. Quiñones, F. J. Cazorla. *Bus Designs for Time-Probabilistic Multicore Processors.* In The Design, Automation, and Test in Europe (DATE) Conference, Dresden, Germany, March 2014.

- M. Slijepcevic, **L. Kosmidis**, J. Abella, E. Quiñones and F. J. Cazorla. *Timing Verification of Fault-Tolerant Chips for Safety-Critical Applications in Harsh Environments.* In IEEE Micro (Special Series on Harsh Chips), Volume 34, Issue 6, p8-19, 2014.

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

- F. Wartel, **L. Kosmidis**, C. Lo, B. Triquet, E. Quiñones, J. Abella, A. Gogonel, and A. Baldovin. *Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study.* 8th IEEE International Symposium on Industrial Embedded Systems (SIES), June 2013.

- L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, **L. Kosmidis**, J. Abella, E. Mezzetti, E. Quiñones, and F. Cazorla. *Measurement-based probabilistic timing analysis for multi-path programs.* In 24th Euromicro Conference on Real-Time Systems (ECRTS), Pisa, Italy, July 2012.

- F. J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, **L. Kosmidis**, C. Lo, and D. Maxim. *PROARTIS: Probabilistically Analysable Real-Time Systems.* ACM Transactions on Embedded Computing Systems (ACM-TECS), Volume 12, Issue 2s, 2013

Chapter 2

Background

Significant efforts have been devoted by industry and academia to devise methods, and the corresponding tools, to verify whether real-time systems fulfil their timing constraints. The increasing complexity of the systems' components (both hardware and software) has driven industry and academia towards increasingly sophisticated timing analysis solutions [Heckmann *et al.* (2003)] [Wilhelm *et al.* (2008)] able to derive tight and reliable WCET estimates in the presence of such complexity. Yet, current sophisticated solutions find difficulties providing tight and trustworthy WCET bounds on modern platforms. This is particularly true for large real-time applications running on top of processors with advanced hardware features (e.g. complex cache memories and multi-core processors) which remains as an open challenge [Abella *et al.* (2015)].

In this Chapter, we review the basic concepts and the main works in the literature regarding the state of the art in the real-time systems domain related to this Thesis. In the first part, we discuss the concept of timing analysis as well as the main families of techniques to perform it. In addition, we introduce the foundations of the timing analysis method which has been the basis of this Thesis, the Probabilistic Timing Analysis. We specifically focus on its Measurement-Based variant (MBPTA), a result of our collaborative effort with several people and institutions inside the PROARTIS project as well as its industrially-viable evolution produced by the PROXIMA project. The second part reviews relevant works in the timing analysis of caches in the real-time domain, both in the deterministic and in the probabilistic analysis domain.

2.1 Timing Analysis

The execution time of a program depends on the environment (execution conditions) in which it is executed. At hardware level, intuitively a given program is

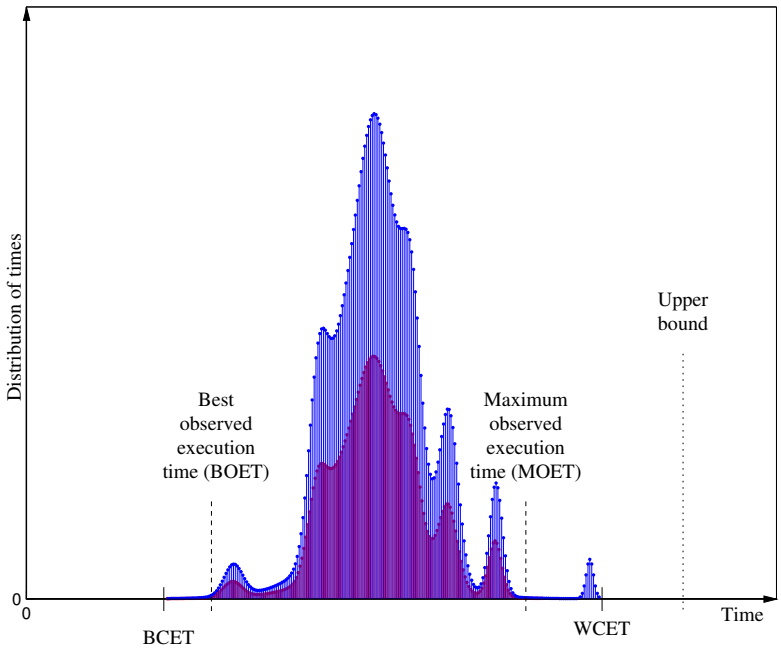


Figure 2.1: Distribution of possible execution times of a task.

expected to have a larger execution time when run on simple hardware than when run on advanced (high-performance) hardware. However, while this rule holds in general, we can find scenarios where programs run slowly on top of advanced hardware. Moreover, there are also cases when a program is executed on modern architectures under the same (theoretical) execution conditions, but its timing behaviour exhibits variability (jitter) due to the hard-to-predict (or undocumented) nature of hardware components.

Another factor adding variability to the execution time is the input set of the application. For instance, program's input data impacts its control flow, i.e. the execution path taken, and hence the execution time of the program. Input also affects the execution time of each individual path. For instance, the latency of some Floating Point (FP) instructions, e.g. division or multiplication, depends on the particular values of the input operands. So when different inputs result in different values operated in the FP units, the same path may have different durations. Finally, the presence of other programs running on the same hardware and sharing the same resources (processor, memory, etc.) also affects its timing behaviour, due to *contention*, also referred to as *interference* in the literature [Rosen *et al.* (2007)] [Pellizzoni *et al.* (2010)] [Dasari *et al.* (2011)]. Figure 2.1 illustrates this phenomenon. In particular it shows how the execution time of a task running on a specific hardware in isolation can vary. In general, it is not possible to obtain the complete execution time distribution, because an exhaustive exploration of all the possible factors that have an impact on the execution time would be required.

However, we can distinguish some important parts of this distribution. In the left end there is the minimum execution time of the task, also known as Best-Case Execution Time (BCET). In the other extreme, there is the maximum execution time, called Worst-Case Execution Time (WCET) [Wilhelm *et al.* (2008)]. These extremes are hard to predict, in general, by running the task in a naive way because in the edges of the execution time distribution, the chance of occurrence is very low. Instead, by observing the execution time we can find a minimum and a maximum observed execution times, which are termed Best Observed Execution Time (BOET) and Maximum Observed Execution Time (MOET) respectively. Unfortunately, there is not an easy way to find out how close to the actual limits these values are.

The execution time distribution lies between the BCET and the WCET. In high-performance computing the main objective is to speed up the execution of frequent events [Hennessy & Patterson (2007)], in order to move the central part of the distribution to the left so that the average execution time is reduced, even at the expense of increasing the WCET if this happens rarely. A prominent example of this type is the speculative execution due to branch prediction, which requires a misprediction penalty to be paid for each wrong speculative event. If the prediction accuracy is high, the central part of the distribution is moved on the left, however the WCET is increased and no guarantees about the prediction accuracy can be provided so that it will not be reached. Conversely, in real-time systems the critical execution time is the WCET (though average case performance is also important), since it determines the time budget that must be granted to a task. For this reason, research on this area has two goals: decreasing the WCET and providing accurate bounds to the WCET. The former aims to improve the worst-case performance of the system, so more complex functionality could be added to the system while still having strong guarantees that all tasks will finish their execution in time. The latter refers to the computation of accurate WCET estimates, i.e. trustworthy and tight: when the WCET estimate is lower than the actual WCET it means that it may result in timing violations. Conversely, if the WCET estimate is largely above the actual WCET it means that we will allocate more resources than needed to run the application, thus leading to an expensive over-provisioned system.

The WCET estimate for a piece of software is computed by a process called *timing analysis*. Currently, there are two main families of timing analysis methods used by industry, the *static* and the *measurement-based* ones [Wilhelm *et al.* (2008)]. The analysis methods can also be classified into *deterministic* and *probabilistic* ones, depending whether they are applied in time-deterministic or time-randomised systems. Independently of the category, all methods try to estimate trustworthy and tight WCET bounds. This requires that the assumptions made by the timing analysis method can be satisfied [Abella *et al.* (2015)]. Moreover,

each method comes with an associated cost for its application in both time and effort, which needs to be below the budget that the industrial users are willing or can afford to spend for each system. In this respect, no timing analysis method is absolutely trustworthy or superior over the others. Industrial users are aware of benefits/costs of each technique and for this reason they do not use a single timing analysis technique for all their developed systems, but they select the appropriate one that better fits the characteristics, requirements and budget of each system.

It is worth clarifying that a task overrun (timing failure) does not necessarily cause a failure in the system behaviour, which would mean a bad-designed safety design [Abella *et al.* (2015)]. Instead, a safety process is followed defining safety goals and a safety strategy to mitigate the risk that hardware or software misbehaviour causes a system fault [RTCA and EUROCAE (1992)] [International Organization for Standardization (2009)]. As the criticality of the task under analysis increases, more mechanisms are put in place (replication, online monitoring, watchdog) to detect and react to undesired situations. The same principle is followed for hardware faults: it is assumed that despite processors have fault detection and correction mechanisms, there is a risk they fail. Hence, mechanisms to react to hardware faults are needed. High-quality WCET estimates are needed to favour the system to continuously operate in normal (non-error) mode and the design of a safe system.

In the following, we introduce each of the analysis techniques, together with their assumptions, which can affect their trustworthiness.

2.1.1 Static Deterministic Timing Analysis

Static refers to the fact that the WCET is computed statically by analysing how the program's would behave in its time domain on the target platform, without actually executing it [Cousot & Cousot (2004)].

Static Deterministic Timing analysis (SDTA) is built on top of mathematical foundations in order to provide strong guarantees over the single WCET estimate it computes. SDTA has been applied in certain high-criticality systems. The trustworthiness of its derived estimate heavily depends on the satisfaction of the conditions under which it is assumed to work. While these conditions can be shown to hold on simple hardware designs and small scale software used in current certified real time systems, the complexity required in both hardware and software to provide the performance needs of future real-time systems, makes these condition hard to satisfy, if at all possible [Mezzetti & Vardanega (2011b)] [Abella *et al.* (2015)]. SDTA involves several steps [Wilhelm *et al.* (2008)] mainly classified in *low-level* and *high-level* analyses. In general, the former deals with the details of the processor architecture while the latter with the behaviour of the program.

The low-level analysis, also found in the literature under the name *Processor-Behaviour* Analysis, is heavily based on the construction of an accurate timing model of the processor architecture under analysis. The possible hardware states of programs are modelled by means of *abstract interpretation* [Cousot & Cousot (2004)] in order to compute a WCET estimate. Interestingly, the trustworthiness of SDTA techniques is limited to the accuracy and the correctness of the timing model. However, hardware vendors are usually reluctant to release a detailed information about their designs, since this possibly reveals their internal structure and could be exploited by the competitors. Even in the case that this information is available, it is frequently scattered in lengthy documents. Moreover, due to the complexity of the hardware design and validation process [Abella *et al.* (2015)], their trustworthiness is questionable as these documents are accompanied by several versions of errata documents, which are updated as soon as a new inaccuracy is found. In fact, the latter is the same obstacle faced by the hardware industry in the adoption of formal verification methods, despite their strong mathematical guarantees for the correctness of a hardware design. As a result of these difficulties, latest works on SDTA rely on measurements on actual hardware [Nowotsch *et al.* (2014)]. However this process requires significant effort from the tool provider since the hardware cannot be analysed only by measurements using a black box approach. Finally, the construction of an accurate model for complex designs beyond the 8-bit and 16-bit micro-controllers traditionally used in the CRTES domain, challenges the scalability of this solution [Heckmann *et al.* (2003)]. For this reason, the static timing analysis community defined design principles that would ease the computation of WCET [Thiele & Wilhelm (2004)] [Axer *et al.* (2014)].

The High-Level analysis studies the program structure using its *control-flow graph* (CFG). The different paths that can be taken at runtime are derived with Implicit Path Enumeration (IPET) [Li & Malik (1995)] techniques. Furthermore, the *value analysis* [Thesing *et al.* (2003)] step is used in order to find infeasible paths and resolve addresses of memory accesses. While value analysis can be used in order to derive statically some actual data values or ranges which will take place at program execution, the dynamic nature of programs makes this infeasible for all program values. For this reason the path analysis is also based on *flow-facts* [Kirner & Puschner (2005)] provided in the form of annotations by an experienced programmer. These flow facts describe information about the software structure that cannot be determined automatically such as maximum loop bounds. However, the trustworthiness of this step heavily depends on the experience and the knowledge of the user about the software under analysis. Moreover, this information might be difficult or infeasible to be obtained in large-scale industrial software [Mezzetti & Vardanega (2011b)].

For the above reasons, the static deterministic timing analysis is currently used in certain industrial contexts, where its assumptions can be satisfied. However, with the increase in complexity of both hardware and software required to meet the performance needs of future real-time systems, SDTA will find difficulties to satisfy its assumptions.

2.1.2 Measurement-based Deterministic Timing Analysis

Measurement-based deterministic timing analysis techniques (MBDTA) [Kirner *et al.* (2004)][Wenzel *et al.* (2005b)][Deverge & Puaut (2005)][Williams (2005)] compute WCET bounds with a process similar to profiling, by executing the program under analysis on the target platform. Due to its simple applicability, this is the most used method to date in industry, including certified systems [Wilhelm *et al.* (2008)] [Law & Bate (2016)]. This family of analysis techniques has its own challenges though.

Measurement-based solutions are composed by two distinct stages: a) analysis and b) system operation. The measurements obtained during the analysis phase are used to predict the maximum execution time of the program during the operation phase, and therefore providing a WCET estimate that upperbounds it. In order this to be effective, the execution conditions at analysis must be as close as possible to the actual execution conditions that may occur during system operation, otherwise there is a risk of underestimating the WCET. Hence, this is one of the most challenging parts of measurement-based approaches.

Another challenge of measurement-based analysis lies on the means used to collect measurements. Various software methods can be used for this task, which affect the quality of collected measurements. Software instrumentation, for instance, modifies the code layout and therefore impacts the measured performance in non-obvious ways [Mezzetti & Vardanega (2011a)] [Diaz *et al.* (2016)] in addition to the extra latency added from the instrumented code. Due to the intrusive nature of instrumentation, the current industrial practice is to retain instrumentation code in the deployed system, despite the runtime overhead that introduces. This is an example of a measure taken in order to guarantee that the deployed system operates under the same conditions as the ones used during the analysis stage.

The execution path of a program during a specific run on top of a particular hardware configuration is determined significantly by its *input set*, also known as *input vector*. The effectiveness of measurement-based techniques is based on the identification of the worst-case input, which can lead to traverse the longest possible path leading to the WCET. In general, the worst-case input is difficult to be determined due to the large input space of a program, and because different events triggered by the software interact in non-obvious ways. This process depends on

an expert user who, based on his experience and the knowledge about the application, has to determine which is the input that likely stresses the most the program under study. However, neither it is guaranteed that such input is indeed, the one leading to the WCET, nor skilled users may exist since significant portions of the code used in CRTES is legacy code [Mezzetti & Vardanega (2011b)], and therefore the original developer may not be involved in this process. Moreover, the source code of these applications may not be available and binaries must be analysed, so the identification of the worst-case input set becomes even more challenging. Even if the source code is available and skilled users of the program under analysis work on its timing analysis, they can provide some hints but not absolute answers because of the complexity of the application and the effects of the hardware in the execution time. For instance, users can identify those paths leading to the WCET, but may be unable to produce those initial execution conditions leading to the WCET such as the placement of memory objects leading to the worst-case cache behaviour.

In order to cope with the input vector problem, path coverage is required from the end user [Deverge & Puaut (2005)] and measurements from different executions are recorded. Note that path coverage is slightly different from the code coverage methods used in the domain of functional testing, such as Random Testing, Basic Block Coverage, Condition/Decision Coverage and Modified Condition/Decision Coverage (MC/DC) [Bünte *et al.* (2011)], which, if used instead, may make the WCET be underestimated. The reason for this is that the interaction of the different hardware resources is hard – if at all feasible – to predict, so no true guarantee exists on the confidence of the WCET estimate obtained by those means.

Some other techniques, instead of using end-to-end measurements of the program's execution time, use multiple measurement points. In this case, the control flow graph of the program is used, in order to determine the execution time of each individual part of the program (e.g., basic blocks, functions, etc.). Additionally, this way offers information about the input path that is covered by each input set, and the coverage of the code. This type of timing analysis can obtain very accurate results for simple hardware. However, this variant also suffers from the proper selection of the input sets, the lack of control of the initial execution conditions and the number of the executions as it is the case of end-to-end measurement-based analysis techniques.

Alternative solutions to the worst input problem, include automated test generation methods [Wenzel *et al.* (2005b)] [Williams (2005)] [Law & Bate (2016)]. These methods identify input vectors that either exercise all feasible paths using techniques similar to the IPET employed by static timing analysis, or select a subset of those which lead to high execution times based on genetic or simulated annealing algorithms. The former are subject to scalability limitations with soft-

ware complexity similar to the static analysis, while the latter make the search for the input vector more tractable, with simulated annealing-based solutions being superior than the genetic-based ones.

In Figure 2.1 the lower part of the distribution (red) depicts an example of results obtained by measurements. The maximum execution time which is observed is called *Maximum Observed Execution Time* (MOET) or *High Watermark*. The main problem of measurement-based techniques, is that, in general there is no evidence about how close MOET and WCET are. For instance, as it is shown in Figure 2.1 the maximum observed execution time can be far away from the actual WCET.

For this reason, the MOET is increased by a margin derived based on the experience of the user about the target platform, in order to cover for the uncertainties of the system's timing behaviour. For example, a typical safety margin used in the avionics industry is 20% [Wartel *et al.* (2013)], which has been shown sufficient since no overruns have been observed with this method for decades. However, this margin is not selected based on any scientific method and, most importantly, it does not necessarily hold when the target platform changes. For instance, according to [Fernández *et al.* (2012)] the 20% is insufficient for multicore processors such as the ones designed for the next generation aerospace applications, since the interference between cores can lead to much more significant slowdowns.

From the above discussion it is evident that the effectiveness of measurement-based methods is highly dependent on the selection of the input sets, the initial execution conditions of the program and the number of the experiments that are performed. Unfortunately, this number should be extremely high in order to be able to achieve some meaningful degree of confidence. Even in that case, existing hardware does not have a probabilistic behaviour and therefore, there is no guarantee that with sufficient number of executions during the analysis phase, the events with statistical relevance regarding the WCET will be captured.

Overall, measurement-based deterministic timing analysis techniques, although widely used in industry, have a number of limitations: their trustworthiness depends heavily on)i) the input sets, and (ii) the correspondence of execution conditions and hardware state between the analysis and system operation as well as (iii) on the number of the experiments performed during the analysis phase. However, it is not clear whether these solution are going to scale with the complexity of future hardware and software critical real-time systems which will complicate both the derivation of stressful inputs as well as viable engineering margins.

2.1.3 Probabilistic Timing Analysis

The notion of probabilities in the analysis of real-time systems has appeared in the literature for long time [Abeni & Buttazzo (1998)] [Gardner & Lui (1999)] [Tia

et al. (1995)] [Atlas & Bestavros (1998)]. However, it was not until early 2000s when probabilistic timing analysis was applied to critical systems [Edgar & Burns (2001)] [Bernat *et al.* (2002)]. In general, those works assume a source of randomisation resulting in a probabilistic execution time. This line of work became widely accepted by the real-time community only recently with the publications [Cazorla *et al.* (2013a)] and [Cucu-Grosjean *et al.* (2012)] – the initial outcomes of the PROARTIS and PROXIMA projects – when for the first time, the assumptions and requirements for probabilistic analysis were analysed together with a demonstration of a practical way to be satisfied in practice. In these articles this was a hardware means, a fully-associative cache with random replacement policy. In the next section, we examine in detail those requirements, since the main purpose of this Thesis is to show how they can be achieved in the general case with careful hardware and software design.

Various references exist to probabilistic and stochastic methods, however not all of them are consistent with the probabilistic timing analysis as it is defined by these two seminal works, which is the basis of most recent related works and it is also the one used in this Thesis. [Cazorla *et al.* (2013a)] gives a comprehensive survey of previous work in the area and clarifies the diverse terminology regarding probabilistic timing analysis.

The proposed methods in the literature can be generally classified on statistical methods and pure probabilistic methods. The first class considers the application of statistical methods to observations in order to extract information about the worst-case execution [Edgar & Burns (2001)] [Hansen *et al.* (2009)]. Initially, those methods have been applied only to soft real-time systems [Abeni & Buttazzo (1998)] [Gardner & Lui (1999)] [Tia *et al.* (1995)] [Atlas & Bestavros (1998)]. This allowed to simplify the analysis and scheduling of hard to analyse tasks such as multimedia applications, without affecting the timing guarantees for hard real-time tasks obtained with classical methods. Around a decade later, the concept has been also extended to hard-real time systems, however, the proposed works are based on assumptions of properties that are not satisfied in the general case [Griffin & Burns (2010)].

The second category includes solutions that consider at least one parameter of the system, to be modelled by a random variable, mainly the execution time [Lehoczky (1996)] [Zhu *et al.* (2002)] or the probability of a path to be taken [Liang & Mitra (2008)]. Similarly, the trustworthiness of these solutions depends on the satisfaction of their assumptions [Abella *et al.* (2015)], which is challenging – if at all possible – to achieve in a real-platform [Abella *et al.* (2015)]. For example, the execution times on conventional platforms are not described by random variables, although the observed execution time may experience some variability due to external to the program reasons, e.g. the memory layout [E.Mezzetti *et al.* (2008)]

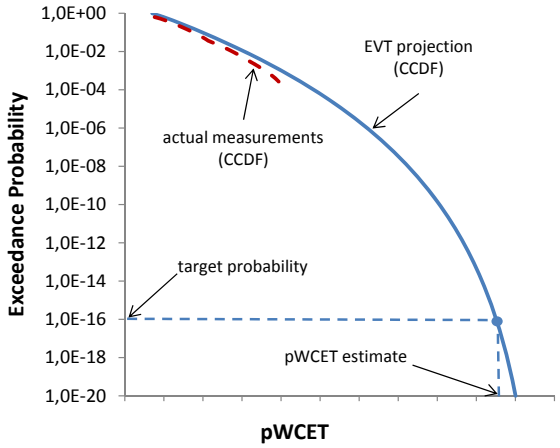


Figure 2.2: Example of the CCDF and tail projection.

[Mytkowicz *et al.* (2009)] or DRAM refreshes [Agirre *et al.* (2016)]. Moreover, even if variability can be probabilistically characterised, not all programs follow the same distributions.

Interestingly, some works take path frequencies (as observed during analysis time) as probabilities. However, in general it is not possible to reason about the frequencies of execution of each path during system operation [Cazorla *et al.* (2013b)].

In the seminal papers [Cazorla *et al.* (2013a)] and [Cucu-Grosjean *et al.* (2012)], we find the first attempts to our knowledge, of making randomisation to emerge from the computing platform, rather than assuming that is provided by user inputs in some way. Both works focus on fully-associative random replacement caches. Although such caches are impractical to be used in real designs with large cache sizes due to the high power dissipation, increased hardware implementation cost and long access latency, these works paved the way for Probabilistic Timing Analysis (PTA), to be considered nowadays as a real alternative to current timing analysis techniques. In this Thesis we contribute in this aspect by proposing solutions in the same line. We propose affordable hardware and software solutions that provide random timing behaviour at platform level.

Similar to the deterministic methods, two variants of Probabilistic Timing Analysis exist: Static PTA (SPTA) [Cazorla *et al.* (2013a)] and Measurement-Based PTA (MBPTA) [Cucu-Grosjean *et al.* (2012)] [Abella *et al.* (2017)]. Although their functioning is different, both methods provide a complementary cumulative distribution function (CCDF), or pWCET function such as the blue curve depicted in Figure 2.2. This distribution upper-bounds the execution time of the program under analysis, guaranteeing that the execution time of a program only exceeds the corresponding execution time bound with a probability lower than a given target probability (e.g., 10^{-15} per run).

It is worth to note that safety standards [RTCA and EUROCAE (1992)] [International Organization for Standardization (2009)] specify maximum probability rates for safety critical systems, with lower probabilities corresponding to higher integrity levels. However, those probabilities refer to hardware failures, which result in functional misbehaviour of the system. Therefore, there is no direct relation of these probabilities to the timing behaviour of the system, neither the target probability selection in PTA. Current software and timing verification on the other hand follows a qualitative approach based on sufficient evidence that the software behaves correctly and timely according to its specification, with an extremely low residual risk, since no method is able to provide absolute confidence [Spanfelner *et al.* (2012)]. PTA on the other hand allows the timing verification process to become quantitative, being able to estimate the residual risk, which in this case is the remaining probability below the cut-off point [Stephenson *et al.* (2013)].

The probabilistic timing behaviour of a program (or an instruction) can be represented with Execution Time Profiles (ETPs). An ETP defines the different execution times of a program (or latencies of an instruction) and its associated probabilities. That is, the timing behaviour of a program/instruction can be defined by the pair of vectors $(\vec{l}, \vec{p}) = \{l_1, l_2, ..., l_k\}\{p_1, p_2, ..., p_k\}$, where p_i is the probability the program/instruction taking latency l_i, with $\sum_{i=1}^{k} p_i = 1$.

The ETP for a program (or instruction) for a given input set leading to a single execution path is obtained as follows for SPTA and MBPTA:

SPTA In SPTA [Cazorla *et al.* (2013a)] [Altmeyer & Davis (2014)] [Davis *et al.* (2013)], ETPs for individual binary-level instructions are determined statically from a model of the processor and software. SPTA is performed by calculating the *convolution* of the discrete probability distributions which describe the execution time for each instruction; this provides a probability distribution, or ETP, representing the timing behaviour of the entire sequence of instructions. For instance the convolution of $ETP_1 = \{\{1, 7\}, \{0.4, 0.6\}\}$ and $ETP_2 = \{\{2, 4\}, \{0.5, 0.5\}\}$ consists in adding the timing vectors and multiplying the probability vectors, resulting in $ETP_3 = \{\{3, 5, 9, 11\}, \{0.2, 0.2, 0.3, 0.3\}\}$. SPTA techniques have been studied for relatively simple processor architectures that use time-randomised caches [Cazorla *et al.* (2013a)] [Altmeyer & Davis (2014)] [Davis *et al.* (2013)] and are not yet close to industry [Abella *et al.* (2015)].

MBPTA Given a set of R runs of a program, on an MBPTA-compliant platform as we explain in Chapter 4, one could compute the pWCET function of the program as the *exceedance cumulative distribution function* (ECDF). ECDF provides the probability of occurrence of each of the observed execution times based on the histogram of execution times. Unfortunately, ECDF can only provide execution time estimates for probabilities down to $\frac{1}{R}$ in the best case. For smaller probabili-

ties, techniques such as *Extreme Value Theory* (EVT) [Kotz & Nadarajah (2000)] [Cucu-Grosjean *et al.* (2012)] are used to project an upper-bound of the tail of the exceedance function, enabling MBPTA techniques to provide pWCET estimates for *target probabilities* largely below $\frac{1}{R}$. Figure 2.2 shows a hypothetical result of applying EVT to a collection of 1,000 observed execution times. The continuous line represents the ECDF function, derived from the observed execution times. The dotted line represents the projection obtained with EVT. The EVT curve can be read as follows: the pWCET estimate which corresponds to 9.5ms for the target probability 10^{-16}, indicates that this execution time can be exceeded with at most that low probability in a given execution of that program.

A difference between SPTA and MBPTA, besides the level of abstraction at which ETPs are to be constructed, is that while SPTA requires ETPs for each instruction to be *determined*, MBPTA simply needs those ETPs for the program or its components (e.g., instructions) to *exist*, but not to be known. ETP may differ for different input sets leading to different execution paths. Each PTA technique has its own methods to combine results from different execution paths. We refer the reader to those methods for further details [Cazorla *et al.* (2013a)] [Cucu-Grosjean *et al.* (2012)]. [Abella *et al.* (2014a)] presents the first comprehensive comparison among STA, SPTA and MBPTA techniques.

The trustworthiness of pWCET estimates obtained with MBPTA on top of random placement caches has been the subject of study in [Reineke (2014)] [Mezzetti *et al.* (2015)]. [Abella *et al.* (2014b)] conducts a thorough analysis of corner cases with the use of MBPTA and proposes ways to address them. [Benedicte *et al.* (2016)] extends the work of [Abella *et al.* (2014b)] and provides two additional and more accurate methods to increase the confidence of MBPTA.

MBPTA is not the only method in the literature to use Extreme Value Theory. In [Yue *et al.* (2011)], the authors use EVT for for time-deterministic architectures to derive execution time bounds. Similarly, [Berezovskyi *et al.* (2014)] [Berezovskyi *et al.* (2016)], apply EVT to measurements collected on a deterministic GPU architecture, in order to derive pWCET of GPGPU kernels. However, the main difference between these works and classical MBPTA ([Cazorla *et al.* (2013a)] [Cucu-Grosjean *et al.* (2012)]) as considered in this Thesis, is that there is no guarantee that the execution conditions at the analysis time upperbound or match the ones at the system operation. In contrast, the use of time randomised architectures, either in software or hardware, facilitates the satisfaction of this property. [Cazorla *et al.* (2013b)] discusses the advantages of using EVT in the context of time-randomised architectures, that is MBPTA, in comparison to applying EVT to time-deterministic architectures. In the same way, [Lima *et al.* (2016)] highlights the importance of using representative inputs at analysis, to those that are used in the deployed system.

MBPTA as conceived in [Cucu-Grosjean *et al.* (2012)] and described in detail in the next subsection, requires independence and identical distribution properties in the collected execution time measurements which are processed by EVT. [Santinelli *et al.* (2014)] relaxes the requirements of using EVT, showing that EVT can be also used under non strict independence scenarios. In particular, a weak dependence or independence of maxima is sufficient. However, in this Thesis we apply EVT only on independent measurements, which are a byproduct of our hardware and software designs as well as how the experiments are performed.

Conventional hardware/software platforms fail to provide the features required by PTA so that pWCET estimates can be computed. Moreover, since measurement-based solutions are closer to industry, the focus of this Thesis is delivering hardware/software platforms whose timing behaviour fulfils the requirements of MBPTA and remove the need for controlling the initial execution conditions, thus reducing the burden on the user to generate those tests leading to the worst-case path.

2.1.4 Introduction to SPTA/MBPTA Requirements

PTA techniques require that the events under analysis, program execution times for MBPTA and instruction latencies for SPTA, can be modelled with *i.i.d.* random variables [Cazorla *et al.* (2013a)]: two random variables are said to be independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other event. Two random variables are said to be identically distributed if they have the same probability distribution.

The existence of an ETP ensures that each potential execution time of the program (for MBPTA) or instruction (for SPTA) has an actual probability of occurrence, which is a sufficient and necessary condition to achieve the desired probabilistic i.i.d. execution time behaviour.

Regardless of whether ETPs are obtained for instructions or full programs, they cannot be derived with current deterministic architectures since events affecting execution time, e.g. cache hits/misses, on those architectures cannot be attached a probability of occurrence. Overall, an SPTA- and MBPTA-analysable system must provide the following properties:

SPTA requires the i.i.d. hypothesis to strictly hold at the granularity level at which ETP are built, i.e. instructions. If the timing probability distribution captured by the ETP of the instruction is fully independent of the execution history, the ETP of the instruction would hold constant across all executions of the instruction. However, this complicates SPTA because different execution paths and even the outcome of the random events in a single path alter the probability vector. SPTA, as presented in [Cazorla *et al.* (2013a)], advocates for doing per-path WCET analysis which are then combined to obtain the WCET estimate

for the program. Within a path, the timing vector of the ETP is insensitive to execution history but the probability vector is not, and therefore, there is a need for bounding probabilistically this dependence.

Since SPTA requires the knowledge of the exact ETP, as discussed in the previous section, its sensitivity in lack of information is equivalent to SDTA techniques [Abella *et al.* (2014a)]. Moreover, the computation of WCET in SPTA requires analytical computations for the convolution at the level of instructions in ETPs for the entire program, therefore its scalability is limited, similar to the SDTA techniques. In this Thesis we focus on the MBPTA variant, which is closer to the industrial needs, due to its simple application process. Therefore, the analytical formulation of ETPs in our proposals are not meant to be exact, that is, it may not be used in the context of SPTA. Instead, they are approximations which show their existence as required by MBPTA.

MBPTA The observed execution times fulfil the i.i.d. property if observations are independent across different runs and a probability can be attached to each potential execution time. To that end, it is enough if we *make the events that may affect the execution time of a program random events*. Hence, taking measurements from a program is equivalent to rolling a dice, with each face having a probability of occurrence. Making enough rolls of the different paths relevant for WCET is enough to apply MBPTA, which derives upper-bounds of the execution time distribution by means of *Extreme Value Theory* (EVT) [Kotz & Nadarajah (2000)][Cucu-Grosjean *et al.* (2012)]. Note that *the existence of the ETPs for each instruction ensures that the execution times are probabilistic and therefore MBPTA can be applied*.

As a measurement-based technique, MBPTA computes a pWCET based on observations collected during the program analysis phase. Therefore, in order the pWCET to be trustworthy, the second fundamental requirement of MBPTA is that *the execution conditions during the analysis must match or upperbound the ones that will take place during operation*. In other words, the potential execution time latencies and their probabilities (ETPs) during the system operation must be guaranteed not to be higher than the ones which have been observed during the analysis [Cazorla *et al.* (2013b)]. As we discuss in Chapter 4, we achieve the execution conditions' upperbounding by forcing the hardware to take its worst case latency at analysis time, or make sure that the same execution conditions exist in both phases by introducing timing randomisation. In the latter case, the hardware will follow a probabilistic timing behaviour modelled by the same ETP during both analysis and operation.

As explained for the SPTA case earlier, ETPs for memory instructions may have dependences. This is also the case for MBPTA, however, due to the second MBPTA requirement, it is enough that those dependences are probabilistic, so

that the measurements (execution times) obtained by running the program probabilistically capture the effect of such dependence.

In this Thesis we focus on MBPTA because it is closer to industrial practice. In Chapter 4 we examine in detail the requirements of MBPTA and how they can be satisfied by a hardware design.

2.2 Caches in Real-Time Systems

Caches are undoubtedly one of the resources with the highest performance impact in a processor system. Most high performance processors come equipped with two levels of cache or even three like the ORACLE SPARC T5, the IBM POWER8 or the Intel Xeon and Core i7. This is also the case of some processors used in the real-time domain such as the ARM Cortex A9 and A15 [ARM Ltd. (2013)], the Freescale P4080 [Freescale Semiconductors (2008)] and the Cobham Gaisler NGMP [Cobham Gaisler (2011)].

Cache memories also impact noticeably the worst-case execution time and have been object of intense study during the last decades by the real-time community. This has motivated researchers to develop models that allow to derive the behaviour of the cache [Mueller & Harmon (1993)] [Ferdinand & Wilhelm (1999)] [Mueller (2000)] [Ferdinand *et al.* (2001)] [Lesage *et al.* (2009)] [Hardy & Puaut (2008)] to determine whether cache accesses hit or miss.

Cache analysis [Ferdinand & Wilhelm (1999)] has been used in the context of static timing analysis to predict the WCET of software by analysing its cache behaviour. For each particular access, its outcome is predicted based on whether it can be ensured to be in the cache (*must analysis*), may be in the cache (*may analysis*) or not evicted after it has been loaded (*persistence analysis*). Those models have been particularly successful for the instruction single-level caches, however, the data cache in the general case remains a major challenge for static WCET analysis methods due to the difficulty of statically determining the address of each data memory access at run-time. This difficulty compels analysis techniques to make pessimistic assumptions, which in turn result in pessimistic WCET estimates. Multi-level caches aggravate these problems; in fact, to our knowledge, few works deal with multi-level caches and only particular setups of multi-level instruction [Hardy & Puaut (2008)] and data caches [Lesage *et al.* (2009)] have been considered so far, evidencing the dimension of the challenge.

The predictability of replacement policies has been characterised by [Reineke *et al.* (2007)]. In particular several replacement policies have been studied and two metrics have been proposed, *evict* and *fill*, to describe how quickly a cache converges to a statically predicted known state.

[Quiñones *et al.* (2009)] proposed non-deterministic caches in real-time systems,

to reduce the chance of pathological cases [Bernat *et al.* (2007)] due to cache risk patterns [E.Mezzetti *et al.* (2008)] in real-time systems.

While static timing analysis techniques demand caches that are deterministic in their temporal behaviour, the introduction of Probabilistic Timing Analysis techniques [Cazorla *et al.* (2013a)] [Cucu-Grosjean *et al.* (2012)] changes the requirements to be accomplished by caches. In particular, PTA requires caches to have a *time-randomised behaviour that make memory accesses to have a probabilistic behaviour* and be assigned an ETP.

In addition to caches, another type of local memory found in some systems is the scratchpad [Banakar *et al.* (2002)]. Examples of processors using scratchpads are the Emotion Engine [Oka & Suzuoki (1999)] chip used in Sony's Playstation 2, the Cell processor [Riley *et al.* (2007)] used in Sony's Playstation 3 and IBM's Blade Servers and modern NVIDIA graphics cards [NVIDIA (2009)]. In the real-time domain, several microcontrollers and processors for the automotive market are based on scratchpads such as the AURIX Tricore [Infineon (2012)]. Scratchpads are software-managed hardware structures, which are used to obtain full predictability in the local memory accesses. However their flexibility is reduced, since they move the burden to the programmer or the compiler for managing the transfers between the main memory and the scratchpads and also to make efficient use of them [Alvarez *et al.* (2015a)] [Alvarez *et al.* (2015b)]. For this reason, all the aforementioned processors use a memory hierarchy that is not based exclusively on scratchpads but contains caches, too.

Scratchpads have been also studied in the literature for real-time systems [Suhendra *et al.* (2005)] [Suhendra *et al.* (2006)] [Kim *et al.* (2014)] [Kim *et al.* (2016)]. Most works propose algorithms to select which parts of code/data should be placed in scratchpads in order to minimise WCET, typically following a set of ILP (Integer Linear Programming) formulations. However this problem is NP-complete, since it is equivalent to the Knapsack problem [Garey & Johnson (1990)].

In order to offer more predictability to caches, cache locking has been implemented in several processors. In an attempt to reduce WCET, several algorithms have been proposed for code [Puaut & Decotigny (2002)] or data cache locking [Vera *et al.* (2003)]. These solutions though, similarly to the scratchpads face difficulties to scale with big program sizes [E.Mezzetti *et al.* (2008)][Mezzetti & Vardanega (2011b)].

Chapter 3

Experimental Setup

3.1 Simulation Framework

Measurement Probabilistic Timing Analysis imposes requirements on the hardware design, which are not found in any currently available processor. We have proposed several hardware designs compatible with these principles which have been introduced in the Background and we extend to the entire processor architecture in Chapter 4. We have evaluated them based on simulation: an essential tool for research development and evaluation, especially in new unexplored areas.

In addition to our simulation infrastructure, we use a set of benchmarks to perform the evaluation of the proposed cache designs. We have used two families of benchmarks that are widely used in the real-time community, Mälardalen [Gustafsson *et al.* (2010)] and EEMBC [Poovey (2007)].

3.1.1 Simulator Description

We use the SystemC component library SoCLib [Pouillon *et al.* (2009)] as a baseline platform in order to create a cycle accurate execution-driven simulator that fulfils our needs. We developed an emulator for a PowerPC 750 processor core, a processor used in current avionics systems [Wartel *et al.* (2013)] based on the model of PowerPC 405 embedded processor provided by SoCLib, which we enhanced with a Floating Point Unit (FPU) and a Memory Management Unit (MMU). Moreover, we implemented a timing simulator modelling an in-order 4-stage pipeline (fetch, decode, execute and write-back), which is compliant with the embedded processors used in several existing CRTES.

The hardware designs and the analysis methods we propose in this Thesis are based on the cache organisations (random placement, random replacement) we introduce in Chapter 5. We created a custom, fully configurable and flexible cache component in our hardware simulator, able to model a system with an arbitrary

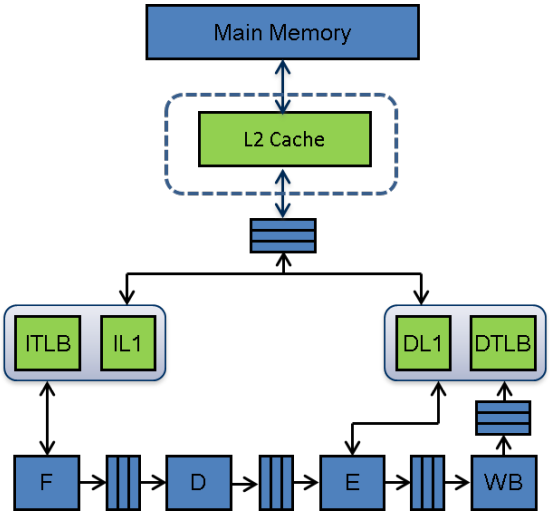


Figure 3.1: General Organisation of modelled architecture. The dashed component (L2 cache) can be disabled.

Table 3.1: Simulator configuration

Processor Configuration	
Pipeline stages	4, in-order instruction execution
Branch Prediction	no, stall until branch resolved
Floating Point Unit	yes
Memory Management Unit	yes
Memory Configuration	
Caches	L1 Instruction and Data caches, L2 split and Unified
TLBs	Instruction and Data
Placement policies	Modulo, Random-Placement
Replacement policies	LRU, Random Replacement
Inclusivity	Inclusive, No-inclusive
Write Policy	Write-Through, Write-Back
MSHR	32 entries
Cache control	Locking, Disabling
Main Memory latency	100 cycles access

number of cache levels using different placement, replacement and inclusivity configurations, as well as with configurable cache hit/miss latencies. Moreover, the same cache model has been used for modelling other cache-like structures such as TLBs.

Figure 4.3 depicts a general model of the architecture we model. As shown in Table 3.1, the cache configuration is completely flexible. Depending on the pro-

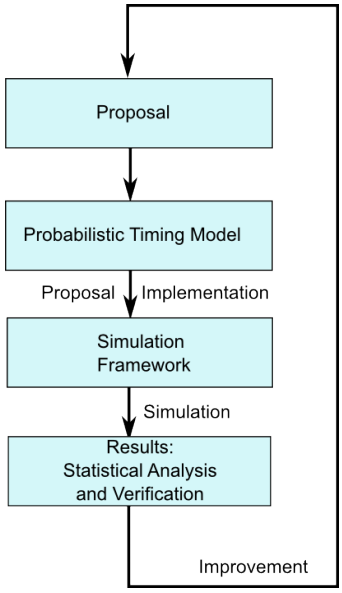


Figure 3.2: Simulation methodology

posal under evaluation, the L2 cache can be disabled completely or when present can be configured as unified or split, and be inclusive of L1 caches. We refer the reader to the Experimental Evaluation section of each Chapter for the actual configuration parameters of the simulator.

Table 3.1 summarises the capabilities of our cache simulator. Caches are indexed with virtual addresses and tagged with physical ones. TLBs are accessed in parallel with caches and in case of miss the pipeline is stalled until it is resolved.

3.1.2 Simulation Methodology

The approach we follow in order to achieve our goals is based on the methodology shown in Figure 3.2. We go through a cyclic refinement process while performing a high level design space exploration. We use a complete simulation framework composed of a processor simulator and a set of benchmarks where we can implement and evaluate our proposals.

For each proposal we first characterise its timing behaviour with a set of analytical mathematical expressions which are based on the probability of each instruction and its latencies, called Execution Time Profile (ETP) as introduced in the Background Chapter. In this way, we show a priori that our proposed system is compliant with Probabilistic Timing Analysis by construction.

Next the proposal is evaluated in our simulation framework, to demonstrate that our mathematical model holds. This is performed by using Measurement-

Based Probabilistic Timing Analysis (MBPTA) [Cucu-Grosjean *et al.* (2012)]. Each benchmark is executed a number of times (∼1000) on our simulator and the execution times of independent end-to-end runs in cycles is collected, as shown in Figure 3.3. We verify that the execution time distribution of each benchmark is independent and identically distributed, so MBPTA can be used.

After this is ensured, we compute the pWCET distribution for each benchmark by applying Extreme-Value Theory (EVT) [Feller (1968)]. Finally, after analysing the different pWCET results, we refine our proposal if it is needed, to provide tighter pWCET estimates.

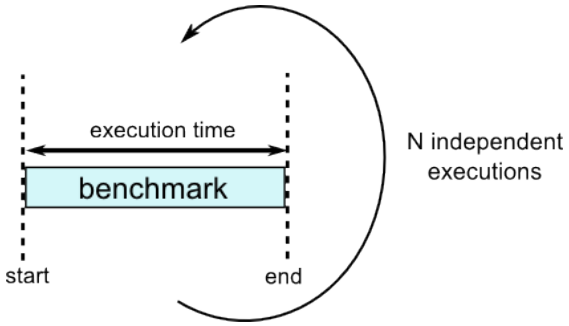


Figure 3.3: Execution time collection for MBPTA

3.2 Metrics

MBPTA-compatible proposals can be verified and evaluated in multiple aspects. First, since MBPTA imposes certain requirements on the platform, those properties need to be verified, before MBPTA can be applied to provide a pWCET curve. Second, the performance of the proposed solution is evaluated.

Requirements. In terms of requirements, as explained in the Background, EVT, the statistical method that MBPTA is based on, requires that its input data, the end-to-end execution times in our case, are independent and identically distributed. As we discuss in the next chapter, our time-randomisation solutions provide this property by design. However, in order to ensure that this property is not violated by mistake in any step throughout our process, we use appropriate statistical tests to validate these properties. In fact, those tests can only indicate the opposite, whether the data are not independent or identically distributed. Moreover, the outcome of these tests is associated with a given confidence level. That is, there is a probability as high as this confidence level, for the tests to report false positives or false negatives.

For independence we use the Wald-Wolfowitz (WW) test [Bradley (1968)]. For identical distribution hypothesis we use the Kolmogorov-Smirnov (KS) goodness-

of-fit test [DeGroot & Schervish (2002)]. We use a 5% significance level (a typical value for this type of tests), whereby absolute values obtained with the WW test must be below 1.96 to prove independence, and the outcome of the KS tests should be above the threshold (0.05) to assert identical distribution.

The second required property for the application of MBPTA is that execution conditions at analysis need to match or upperbound those at operation, in order to avoid experiencing at operation longer latencies than the ones observed during the analysis phase of the system. This property is ensured in our proposals by construction and therefore it is only assessed qualitatively. In particular, as we explain in the next Chapter, for the cache we have opted the timing randomisation option, which ensures that it will follow the same probabilistic timing behaviour in both analysis and operation time, and therefore operate under the same execution conditions. For other jittery resources with lower jitter, such as the floating point unit, we design them to operate in their highest latency during analysis, so that analysis observations are always upperbounding the execution time at operation.

Performance. The performance of each proposal can be assessed either in worst case terms, which is more relevant for CRTES, or in the average case.

For the worst case we use a pWCET estimate derived with MBPTA at the level of cut off probability 10^{-15}. This value provides a sufficiently low residual risk appropriate for use in CRTES, comparable – but not directly related, as discussed in the Background – with the maximum hardware failure rates found in safety standards in avionics [RTCA and EUROCAE (1992)], where the highest criticality software should have a maximum failure rate of 10^{-9} per hour of operation. In the case that an equivalent MBPTA compatible solution is available, i.e. when we improve over our previous proposals, the corresponding pWCET at this probability level for both proposals is used for the comparison. It is worth to note however, that we do not provide any comparison with other timing analysis methods, since this task has been performed in [Abella *et al.* (2014a)], where it is shown that a fair comparison can only be performed with assumptions or on a platform not being the best fit for any of the methods.

Finally, while the average performance is not the primary goal in CRTES, it is still important to be known. Average performance is a key metric of interest in high-performance computing and therefore, showing that MBPTA solutions do not penalise significantly the average performance is very important for adoption of our proposals in those platforms. Moreover, average performance is important in the CRTES domain, in order to know whether the pWCET estimate is close to the average execution time, and therefore it is not overestimated significantly. For this reason, we also include average performance results for our proposals, which are computed as the arithmetic mean of the collected execution times.

3.3 Benchmarks

The benchmarks we use come from two big families which are commonly used in the CRTES domain, EEMBC and Mälardalen. The EEMBC automotive suite [Poovey (2007)] is developed by the Embedded Microprocessor Benchmark Consortium and is specially designed to measure performance of embedded processors by executing common tasks performed in an automotive system as control of gear rotation, ignition system, etc. These benchmarks have the following structure: they are composed of a main loop, which is executed a high number of iterations depending on the benchmark, with few function calls in its body. The input data are embedded in the application emulating memory mapped sensor data, so during each iteration a different value is processed.

The other benchmark suite that we use is the Mälardalen one [Gustafsson *et al.* (2010)], which consists of a set of open-source benchmarks. Similarly to EEMBC, each application comes with an embedded input set, and has simple structure with mostly linear code and a few small loops. Due to this fact, is one of the most common choices for timing analysis techniques. In Table 3.2 we present a short description of each benchmark in each benchmark suite.

For each benchmark we collect the execution time of a number of end-to-end runs, which is usually 1000, as shown in Figure 3.3.

Table 3.2: Benchmarks used in our simulation environment

EEMBC Autobench	
a2time	Angle to Time Conversion
basefp	Basic Integer and Floating Point
bitmnp	Bit Manipulation
cacheb	Cache "Buster"
canrdr	CAN Remote Data Request
aifft	Fast Fourier Transform (FFT)
aifirf	Finite Impulse Response (FIR) Filter
aiifft	Inverse Fast Fourier Transform (iFFT)
aiirflt	Infinite Impulse Response (IIR) Filter
matrix	Matrix Arithmetic
pntrch	Pointer Chasing
puwmod	Pulse Width Modulation (PWM)
rspeed	Road Speed Calculation
tblook	Table Lookup and Interpolation
ttsprk	Tooth to Spark
Mälardalen benchmarks	
adpcm	Adaptive pulse code modulation algorithm
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
edn	Finite Impulse Response (FIR) filter calculations.
expint	Series expansion for computing an exponential integral function.
fac	Calculates the faculty function.
fdct	Fast Discrete Cosine Transform.
fft1	1024-point Fast Fourier Transform using the Cooly-Turkey algorithm.
fibcall	Simple iterative Fibonacci calculation, used to calculate fib(30).
fir	Finite impulse response filter over a 700 items long sample.
insertsort	Insertion sort on a reversed array of size 10.
janne_complex	Nested loop program.
jfdctint	Discrete-cosine transformation on a 8x8 pixel block.
lcdnum	Read ten values, output half to LCD.
lms	LMS adaptive signal enhancement.
ludcmp	LU decomposition algorithm.
matmult	Matrix multiplication of two 20x20 matrices.
minver	Inversion of floating point matrix.
ndes	Bit manipulation, shifts, array and matrix calculations.
ns	Search in a multi-dimensional array.
nsichneu	Simulate an extended Petri Net.
prime	Calculates whether numbers are prime.
qsort-exam	Non-recursive version of quick sort algorithm.
qurt	Root computation of quadratic equations.
recursion	A simple example of recursive code.
select	Selection of the Nth largest number in a floating point array.
sqrt	Square root function implemented by Taylor series.
st	Statistics program.
statemate	STAtechart Real-time-Code generated code
ud	Calculation of matrixes.

Chapter 4

MBPTA-Compatible Processor Design

4.1 Introduction

Measurement-Based Probabilistic Timing Analysis (MBPTA) methods [Cazorla *et al.* (2013a); Cucu-Grosjean *et al.* (2012); Hansen *et al.* (2009)], as explained in the Introduction and Background chapters, allow the execution time of applications to be accurately modelled – *at some level of execution granularity* – by a probability distribution. Under MBPTA, at a given level of granularity of execution, the response time of each execution component (e.g. an instruction) at that level is assigned a distinct probability of occurrence. This trait is described by a probabilistic Execution Time Profile (ETP), which is expressed by the pair: <timing vector; probability vector>. The timing vector in the ETP of an execution component enumerates all its possible response times. For each response time in the timing vector, the probability vector lists the associated probability of occurrence. Hence, for execution component $\mathcal{C}_i$ we have $ETP(\mathcal{C}_i) = <\vec{t_i}, \vec{p_i}>$ where $\vec{t_i} = (t_i^1, t_i^2, ..., t_i^{N_i})$ and $\vec{p_i} = (p_i^1, p_i^2, ..., p_i^{N_i})$, with $\sum_{j=1}^{N_i} p_i^j = 1$.

The processor architecture may facilitate ensuring that individual instructions have an associated ETP. As this feature in turn enables a sound and effective application of MBPTA, this is the level of execution granularity at which we concentrate in this Chapter.

Contribution. In this Chapter, we describe the architecture features that a processor should possess to guarantee MBPTA fitness by construction (also termed *MBPTA-friendliness* or *MBPTA-compliance*) and we also offer insight on the costs that may be incurred in actual implementation. To that end we categorise processor resources according to their timing behaviour and detail how they should be designed so that they can be used in a MBPTA-friendly processor. Without loss

of generality, we consider the inner operation of the processor to employ a number of passive resources (e.g. cache, buffers, buses, etc). We assume each processor instruction to use some of those resources in a given order, whether in sequence or in parallel. We design processor resources so that each can be assigned a given ETP. To achieve this for all resources, we use *time randomisation* in *some*, actually very few, of them. Resources that are not time-randomised must be assigned a local upper bound to their response time that can be safely composed.

Given the relatively simple processors used in CRTES, and since MBPTA compliance is introduced for the first time, we focus on single-core in-order architectures employing cache structures as main performance enhancing feature. More sophisticated architectures found in mainstream and high-performance computing such as multi-cores, many cores, General Purpose Graphics Processing Units (GPGPU), Non-Uniform Memory Access (NUCA) hierarchies etc., are not covered. Note however, that CRTES industry is very conservative in the use of new features, e.g. in many systems caches are disabled for "safe" operation [Rebaudengo *et al.* (2003)] [Santini *et al.* (2014)]. Further note that the same design principles defined in this Chapter can be applied to construct MBPTA-compatible processors with those features, as it is already the case of follow-up studies of this Thesis regarding multi-cores – briefly discussed later in this Chapter.

4.2 Requirements on Hardware Design

PTA, as understood in this Thesis, can be applied in either a static (SPTA) [Cazorla *et al.* (2013a)] or measurement-based (MBPTA) [Cucu-Grosjean *et al.* (2012)] fashion. In this Chapter, as well as in this Thesis, we focus on the measurement-based variant of PTA. We refer the reader to the Chapter 2 for the basics of PTA.

MBPTA considers events resulting from the observation of end-to-end measurement runs of the program, thus at coarser granularity than processor instructions. MBPTA uses a statistical method called Extreme Value Theory (EVT) [Kotz & Nadarajah (2000)][Cucu-Grosjean *et al.* (2012)] to compute pWCET estimates and therefore needs the observed execution times to describe *probabilistically independent* events. EVT requires that the observed execution times of program runs (of the same path) have a distinct probability of occurrence and can be modelled with independent random variables. The MBPTA method described in this Thesis also causes those variables to be identically distributed, so that MBPTA treats i.i.d. random variables[1]. In addition to this property, unlike SPTA which only re-

[1]Two random variables are said to be independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other event. Two random variables are said to be identically distributed if they have the same probability distribution. Unfortunately, these properties are not met by current processors due to their

quires i.i.d., MBPTA places strong requirements on how the observations are made (which of the many possible runs are considered) and what they can observe of all possible outcomes (which execution paths they cover). In particular, MBPTA requires that the execution conditions under which the observations are collected, to match or be worse than those that will take place at system operation.

The axiomatic existence of an ETP per *dynamic* instruction (by which we mean an individual instance of that program instruction in a given run of the program) ensures that, under MBPTA, each potential execution time of the program has a distinct probability of occurrence, that is every program run has an associated ETP, which is a sufficient and necessary condition to achieve the prerequisite i.i.d. execution time behaviour [Abella *et al.* (2013)].

Unfortunately, regardless of whether ETP are sought for program instructions or full programs, they *cannot* be granted with current processor architectures since the events that affect their execution time, e.g. cache hits/misses, cannot be attached a probability of occurrence. So we need to set out to understand what features a processor architecture should possess to allow ETP to exist.

4.3 Modelling the Timing Behaviour of Processor Resources

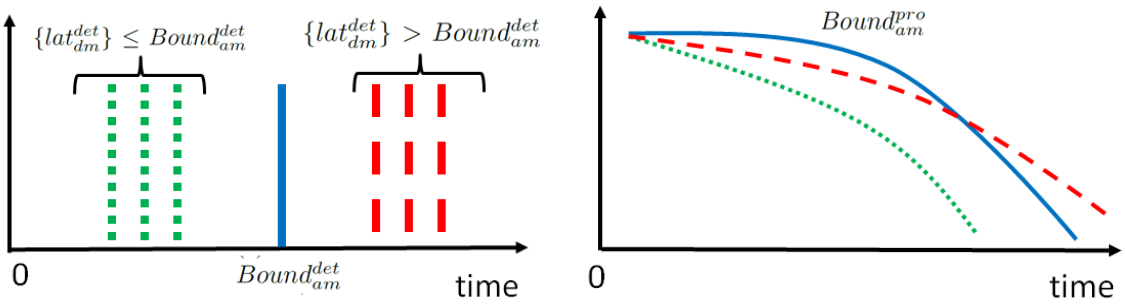
In order to enable the use of MBPTA, the latencies with which each resource responds should have an attached probability of occurrence. The execution time of the instructions using those resources can then also be captured probabilistically. In this respect, the probabilistic execution time of an instruction is a function of the ETP of the resources it uses and how they are arranged, in series or in parallel. Ultimately, this enables capturing the execution of the whole program, which is comprised of instructions, in a probabilistic manner.

For a processor architecture to be MBPTA-friendly, the pWCET estimates obtained for the programs that run on it must hold valid for the whole operational life of the system, hence for every run of the programs of interest under all execution conditions. To understand how the timing behaviour of processor resources needs to be modelled for those guarantees to be obtained, we first need to appreciate how the MBPTA process works.

4.3.1 Analysis and Operation Phases

As a measurement-based technique, MBPTA has two phases: analysis and operation. Execution time observations are collected during analysis in order to make

state-sensitive nature.



(a) deterministic-latency resource (b) probabilistic-latency resource

Figure 4.1: Deterministic and probabilistic upper-bounding latencies

accurate predictions regarding the execution times at system operation. As a consequence, systems amenable to MBPTA have two distinct modes of use: one for the analysis phase, and another for the operation one.

- The analysis mode is used to obtain pWCET estimates that hold valid during system operation. To this end, the timing behaviour of the system in that mode must upper bound that of the system after deployment, as used in real scenarios. This guarantees that any circumstance that can occur during the lifetime of the system cannot alter its timing behaviour in a way that has not already been upper bounded at analysis time.

- The operation mode is used during actual deployment of the final system. In this mode, timing conditions are unrestricted and can thus lead to lower execution times than those experienced in the analysis mode.

4.3.2 Deterministic and Probabilistic Upper-bounding

By intent, the analysis mode requires that the timing behaviour of the system as a whole and of its individual components in isolation (seen at the granularity of execution of interest) either upper bounds or matches that which will occur in operation mode. For MBPTA-friendly processor architectures, this condition can be achieved in either a deterministic or a probabilistic manner. Accordingly, any pWCET estimate obtained by analysis is a trustworthy upper bound of the execution times that may occur after deployment in operation. Next we discuss what needs to be done for different hardware resources.

Figure 4.1 provides a schematic view of the meaning of (a) deterministic upper-bounding and (b) probabilistic upper-bounding. In both sides of the figure, the x-axis represents execution time, and the y-axis the probability for any particular latency to occur (this is obviously 1 in the case of deterministic resources). In Figure 4.1(a), the solid vertical line represents the analysis-mode bound (am),

$Bound^{det}_{am}$ for the latency of a component. If in the operation-mode (dm), the actual latencies, $\{lat^{det}_{dm}\}$, are below $Bound^{det}_{am}$, which is shown with the dotted lines, then the obtained bound is trustworthy. If it cannot be ensured that this is the case, the operation-time actual latencies (dashed lines) can be bigger than the analysis-mode bound $\{lat^{det}_{dm}\} > Bound^{det}_{am}$, hence the bound is not trustworthy and cannot be used. In Figure 4.1(b) the solid curve represents the analysis-mode upper-bound ETP of the latency of the resource, $Bound^{pro}_{am}$. We say that $ETP_i \geq ETP_j$, that is, ETP_i probabilistically upper-bounds ETP_j, if for any cutoff probability the execution time of ETP_i is higher or equal than the execution time of ETP_j. Hence, if actual latencies for the resource are like the dotted curve, then they are probabilistically upper-bounded by $Bound^{pro}_{am}$ (solid line). However, if actual latencies match those described by the dashed curve, they are not probabilistically upper-bounded by $Bound^{pro}_{am}$.

4.3.3 Benefits

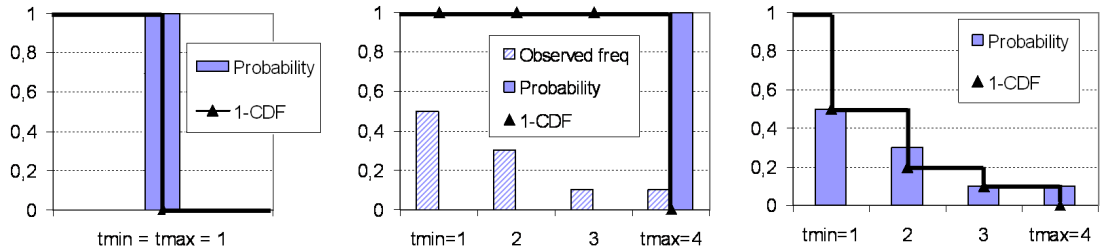
Upper-bounding, in either its deterministic or probabilistic form, allows to ensure that the execution conditions at analysis time, will match or upper-bound the ones at operation. Moreover, it removes the dependence of low-level sources of execution time variation on the input data values, simplifying the analysis [Cazorla *et al.* (2013b)]. This way, the end-user is relieved from the burden to control these sources of execution time variation. In fact, the sole requirement of the analysis process is reduced to provide only sufficient path coverage [Cucu-Grosjean *et al.* (2012)], as opposed to deterministic measurement based timing analysis methods.

4.3.4 Taxonomy of Hardware Resources

We term *jitterless resources* the processor resources that have a fixed latency, independent of the input request and of the past history of service. several hardware resources in current processor architectures are jitterless. Jitterless resources are easy to model for all types of static timing analysis. For MBPTA techniques, the ETP of a jitterless resource jl is given by: $ETP_{jl} =< (l), (1.0) >$, where l is the latency of the resource. Its PDF is shown in Figure 4.2(a).

Other resources, for instance cache memories, have a variable latency: we call them *jittery resources*; their latency depends on their history of service, i.e. the execution history of the program, the input request, or a combination of them. Let us discuss each such case in turn:

- Dependence on execution history. Some resources are stateful and their state is affected by the processing of requests. If latency depends on the internal state of the resource and this state is in turn affected by previous requests,



(a) Jitterless resource (b) Jittery upper-bounded resource (c) Jittery time-randomised resource

Figure 4.2: Probabilistic timing behaviour of a single instruction for each type of resource

then we say that the resource latency depends on the execution history of the program. With caches, the latency of an access request depends on whether the access is a hit or a miss, which in turn depends on the sequence of previous accesses to memory.

- Dependence on input request. The latency is determined by the data carried by the request: for a processor, data are usually encoded in the instruction that issues the request, or stored in its input registers.

Jittery resources have an intrinsically variable impact on the WCET estimate for a given program. The significance of this impact depends on the magnitude of the jitter, the program under study, and the analysis method. For any given jittery resource, either all requests to it are assumed to incur the worst-case latency – as long as timing anomalies can be excluded [Reineke *et al.* (2006)] – or the resource is time-randomised. The design choice for a given resource needs to trade the design cost for time randomising against the degradation of WCET tightness for always assuming worst-case latency.

The ETP for a resource r_{wl}, assumed or configured to worst-case latency, can be expressed as $ETP_{r_{wl}} =< (l_{max}), (1.0) >$, where l_{max} is the worst-case latency of the resource. An example of the impact of such upper bounding is shown in Figure 4.2(b). In the example, the actual probabilities for each latency are unknown; only frequencies can be obtained; upper bounding therefore is needed. Conversely, the ETP of a time-randomised jittery resource r_j is: $ETP_{r_j} =< (l_j^1, l_j^2 ..., l_j^k), (p_j^1, p_j^2, ...p_j^k) >$ where l_j^i and p_j^i represent the different latencies of the resource r_j and their associated probabilities of occurrence. This is shown in Figure 4.2(c).

4.3.5 Assigning ETP to Individual Processor Resources

The probability for a given latency is different from its frequency. This is best shown by an example. Consider a resource R_1 with $\vec{t_1}= (t_1^1, t_1^2)$: latency t_1^1 in the timing vector would have a true probability of occurrence $p_1^1 = 0.5$ if – in the implementation of that resource – on every request to it we tossed a coin and the request had latency t_1^1 if we saw heads and t_1^2 otherwise. In contrast, if for a deterministic stateful resource R_2 with latency $\vec{t_2}= (t_2^1, t_2^2)$ we *observed* that, for a given program, 50% of the requests take t_2^1 and 50% t_2^2, we would have a 50% observed frequency for each possible latency of that resource, but not a true 50% probability. This is so because for events that are strictly dependent on the history of execution, information on past events *cannot* be used to provide guarantees about the appearance of future events.

For the purposes of MBPTA, the timing behaviour of jitterless and jittery (either upper-bounded or time-randomised) resources can all be described probabilistically by ETP.

4.3.6 ETP of several execution components

A composite ETP can easily be determined for every individual program component (ETP_{pc}), e.g. a dynamic instruction, that uses processor resources, which has an associated ETP describing their latency. That is $ETP_{pc} = f\,(ETP_1, ETP_2, ..., ETP_n)$, where ETP_i is the probabilistic execution time of resource r_i.

Sequential composition: sequential composition of ETP, $f_s\,(ETP_1, ETP_2, ..., ETP_n)$, leads to an ETP where latencies and probabilities are determined by the type of dependence across the input ETP (whether systematic or probabilistic, as shown in [Abella *et al.* (2013)]). Sequential composition as intended here is architectural and not mathematical, hence different from the convolution used in the context of SPTA for combining ETPs of static instructions (e.g. instructions in the object code of the program).

Let us assume two ETPs, $ETP_1 =< (1, 2), (0.5, 0.5) >$ and $ETP_2 = < (5, 10), (0.5, 0.5) >$. Further assume that whenever ETP_1 takes latency 1, then $ETP_2 =< (5, 10), (0.8, 0.2) >$ and whenever ETP_1 takes latency 2, then the second ETP is $ETP_2 =< (5, 10), (0.2, 0.8) >$. In this case, $ETP_{1+2} = f_s\,(ETP_1, ETP_2)$, leading to $ETP_{1+2} =< (6, 7, 11, 12), (0.4, 0.1, 0.1, 0.4) >$. Still, ETP_2 takes, for instance, latency 5 with probability 0.5 because $P(ETP_1 = 1) \times P(ETP_2 = 5) + P(ETP_1 = 2) \times P(ETP_2 = 5)$ is $0.5 \times 0.8 + 0.5 \times 0.2 = 0.5$.

The key appreciation here is that the dependence that ETP_2 has on ETP_1 can be modelled probabilistically. As a result, the executions carried out during analysis, capture the behaviour of this dependence and hence, cause it to be covered by the pWCET estimate derived to bound the execution time during operation.

Parallel composition: processor resources may also be arranged in parallel. Examples of parallel resources are some particular designs of cache memories and TLB, where cache access and address translation can occur in parallel. With parallel arrangements, no dependence across ETP can exist, since for that to exist some sequential relation across ETP should occur, which should be addressed by sequential composition. The probabilities of the parallel composition ($f_p\left(ETP_1, ETP_2, ..., ETP_n\right)$) correspond to the multiplication of probabilities across ETP. However, the latencies correspond to the maximum latency of the probabilities multiplied. This is illustrated with the following example. Let the ETP for two program components be $ETP_1 =< (1,4),(0.4,0.6) >$ and $ETP_2 =< (2,3),(0.3,0.7) >$ respectively. The ETP from their parallel composition, $ETP_{1+2} = f_p\left(ETP_1, ETP_2\right)$, is $ETP_{1+2} =< (2,3,4),(0.12,0.28,0.6) >$.

4.3.7 More Complex Single-core Processor Architectures

We have shown that jittery deterministic resources need to be redesigned to make their timing behaviour amenable to MBPTA by construction. This can be done by either randomising their timing behaviour or enforcing them to their worst-case latency. Resources with probabilistic latency perfectly fit the MBPTA principles. However, jittery processor resources exist that do not easily fit in the taxonomy we used in Section 4.3.4. This is the case of resource buffers, also known as FIFO queues or simply buffers.

A buffer resource may stall if it gets full, which increases the latency of the requests that use it. Stalls across pipeline stages may for example occur owing to contention for buffer space; those stalls would be real enough to fear, but difficult to predict causally.

The main characteristic of buffer resources, however, is that they are not *sources of jitter* but rather *jitter propagators*[1] [Kosmidis *et al.* (2013e)]. The intuition here is that if all jitter that occurs in a processor is probabilistic, that is, it is solely due to time-randomised resources, any combination of random events has a given probability of occurrence. Now, as every single combination of events causes the program to incur a distinct execution time, each execution time has a distinct probability of occurrence. For each combination of random events, resource buffers may get full and consequently increase the execution time of the program. However, buffers themselves do *not* introduce any change in the probability distribution of random events. The presence of buffers may well cause the execution time of the program to vary, but each execution time continues to have a true probability of occurrence, which is what MBPTA requires.

[1]This is another contribution of this thesis. However, in order to keep the size of this thesis in reasonable size, we discuss it briefly in the current section instead of a separate chapter.

In general, all hardware resources can be made MBPTA-friendly: it suffices that they either do not introduce jitter on their own (hence they are fixed-latency or else just jitter propagators) or their jitter can be upper-bounded or else it can be randomised. In that manner, initial conditions no longer need to be accounted for as sources of execution time variation [Cazorla *et al.* (2013b)], and trustworthy pWCET estimates can be obtained. Those pWCET bounds can also be tight because MBPTA-friendly architectures cause the build-up effects of abrupt pathologically large variations in execution time (for the same random variable) to naturally smooth out so that pessimism can be much reduced.

4.3.8 First Steps Towards MBPTA-friendly Multi-cores

In multi-core architectures, in addition to all the sources of execution time variability that appear in a single-core architecture, a further one arises: inter-task interference[1]. In single-core architectures, given two instructions i_x and i_y of the same program, where the subscripts determine the order in which each instruction is fetched into the processor, i_y may have a potential impact on the execution time of i_x only if $y < x$, meaning that i_y executes prior to i_x. In a multi-core, when several tasks run in parallel, the execution time of one instruction i_x^{T1} belonging to task T_1 may be affected by any other instruction i_y^{T2} from task T_2. If there is precedence ordering in task execution such that T_2 executes after T_1, then the inter-task interference generated by i_y^{T2} does not affect i_x^{T1}. If no precedence ordering can be asserted instead, T_1 and T_2 can execute in any order. Hence they may execute in parallel on different cores, so that i_y^{T2} may cause inter-task interference on i_x^{T1}. It is evident that we cannot conceivably capture the effect that any single instruction of any task i_j^{Tk} may have on any other instruction of any other task i_l^{Tm} in the system. Should this be required, MBPTA would become intractable. To prevent this, the design of MBPTA-friendly multi-cores must ensure that the worst effect that one task can have on the execution of any other task due to inter-task interference can be probabilistically bounded.

Interestingly, the MBPTA-friendly design principles already outlined for single-core processors extend quite well to the design of multi-core architectures. The resources for which this approach is most advantageous are those that are shared upward the processor hardware architecture off the core, where they may cause massive inter-task interference.

- **Shared bus**. The authors of [Jalle *et al.* (2014)] show that the arbitration latency of a shared bus can either be upper bounded at analysis time

[1]This term does not include the interference that in single core processor occurs in caches and TLBs owing to context switches. This is intentional as this overhead can be quantified probabilistically [Davis *et al.* (2013)].

or randomised so that the timing behaviour observed at analysis matches or upper-bounds that which may emerge during operation. In fact, upper bounding the bus arbitration latency has been shown to be viable also for time-deterministic systems [Paolieri *et al.* (2009a)].

- **Shared memory controller**. The same cited work [Jalle *et al.* (2014)] shows that the latency of a shared memory controller can be upper bounded, which is fine for MBPTA-friendliness. Again, that approach is in line with findings for time-deterministic systems [Paolieri *et al.* (2009b)].

- **Shared cache**. Cache partitioning has been proved to be a practical way to attenuate the interference effects from cache sharing. This solution was first shown for time-deterministic systems [Paolieri *et al.* (2009a)]. However, since it eliminates all cache conflicts among tasks running on different cores, it cancels out the multi-core side of the cache problem, and allows using, for each CPU, the solutions devised for single-core processors.

  An alternative approach has been put forward in [Slijepcevic *et al.* (2014)], where a hardware feature is proposed to limit the eviction frequency caused by individual tasks on a shared time-randomised cache. That mechanism allows controlling inter-task interference without resorting to cache partitioning, which reduces the pWCET against the partitioned case.

4.4 Case Study

4.4.1 Designing an MBPTA-friendly Processor Architecture

We use a 4-stage pipelined core architecture as depicted in Figure 4.3. The four stages operate as follows: Instructions are fetched in-order from the instruction cache (IC) into the processor. The ITLB is accessed in parallel to translate pages. Instructions are decoded in one cycle. Decoded instructions are issued in-order to the execute stage. Instructions are executed in one cycle, except for memory operations. Read operations access the data cache (DC) during this cycle, blocking the execution of further instructions until data are fetched. The DTLB is accessed in parallel with the DC. The instructions commit in one cycle. Write operations update the data cache during this cycle.

IC and DC are 16 KB in size, 8-way set-associative, with 16 bytes per line. The DC and IC use the random placement and replacement policies presented in Chapter 5.The DC is write-through, so that all store instructions are propagated to memory. The DC is write-allocate, hence data are fetched on a store miss.

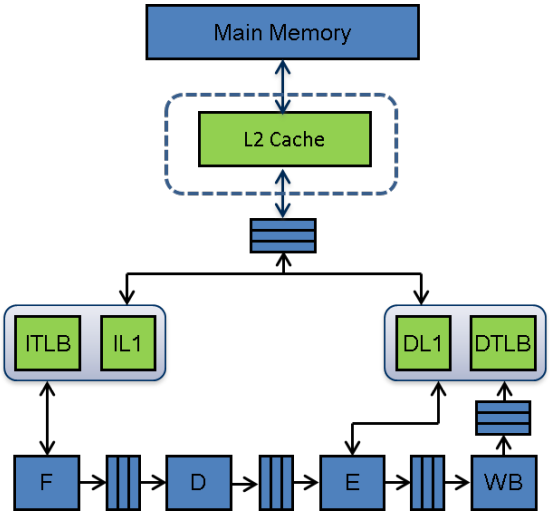


Figure 4.3: Reference core architecture. L2 cache (dashed component) is not used in the case study.

ITLB and DTLB are 8-entry fully-associative random-replacement, with 1 KB page size, and their misses are handled by a hardware page-walker. The hit and miss latencies are 1 and 80 cycles respectively. The TLBs are accessed in parallel with the caches so that instructions are stalled in the corresponding stage until both the cache *and* the TLB can serve the request.

4.4.2 Deriving ETP

In the example architecture, there are two main sources of execution time variability: TLB and caches, which inject random events in the execution time of a program.

We differentiate between two types of instructions: those that operate on the core (e.g. add, div, mult); and those that operate on memory (e.g. load, store). Core operations take a variable latency depending on whether they hit in the instruction cache and instruction TLB, whose ETP (ETP_{IC} and ETP_{ITLB} respectively) are composed in parallel, and memory latency, which is accessed in case of a miss and whose ETP (ETP_{IM}) is composed sequentially with the composition of the instruction cache and the instruction TLB. This leads to what we term the ETP of the front-end (`fend`): $ETP_{fend} = f_s\left(f_p\left(ETP_{IC}, ETP_{ITLB}\right), ETP_{IM}\right)$. Then, the resulting ETP, ETP_{fend} needs to be composed with the ETP of the buffer between the front-end and the back-end, ETP_{buf1}, the ETP for core operations, ETP_{exec}, the ETP of the buffer after execution, ETP_{buf2}, and the ETP of the write-back stage, ETP_{wb}. While ETP_{exec} and ETP_{wb} have the form $< (l), (1.0) >$,

ETP for buffers have as many latencies as potential stalls they may produce, and their probability vector is 0.0 for all latencies but one, whose probability is 1.0. Which latency has probability 1.0 is determined by the state left by previous instructions. More details about how buffers increase execution time without expanding the number of probabilistic states can be found in [Kosmidis *et al.* (2013e)]. Since all those actions occur sequentially, the ETP for core operations is as follows:

$$\begin{aligned} ETP_{core} \;=\; & f_s\left(ETP_{fend}, ETP_{buf1},\right. \\ & \left. ETP_{exec}, ETP_{buf2}, ETP_{wb}\right) \end{aligned} \tag{4.1}$$

Memory operations have the same front-end ETP as core operations. The back-end latency, instead of depending on ETP_{exec}, depends on the time of the data memory path (`dmpath`) composed by the data cache and the data TLB, which are accessed in parallel, and memory latency, which is accessed sequentially: $ETP_{dmpath} = f_s\left(f_p\left(ETP_{DC}, , ETP_{DTLB}\right), ETP_{DM}\right)$. Therefore, the ETP for memory operations is as follows:

$$\begin{aligned} ETP_{mem} \;=\; & f_s\left(ETP_{fend}, ETP_{buf1},\right. \\ & \left. ETP_{dmpath}, ETP_{buf2}, ETP_{wb}\right) \end{aligned} \tag{4.2}$$

4.4.3 Checking the i.i.d. Hypothesis

The existence of an ETP for individual instructions ensures that the program execution times exhibit the prerequisite i.i.d. property of MBPTA. With MBPTA, we empirically ascertain whether this claim holds, by using proper i.i.d. tests applied on the execution times of running EEMBC benchmarks [Poovey (2007)] on the processor architecture.

For independence we use the Wald-Wolfowitz (WW) test [Bradley (1968)]. For identical distribution hypothesis we use the Kolmogorov-Smirnov (KS) goodness-of-fit test [DeGroot & Schervish (2002)]. We use a 5% significance level (a typical value for this type of tests), whereby absolute values obtained with the WW test must be below 1.96 to prove independence, and the outcome of the KS tests should be above the threshold (0.05) to assert identical distribution.

For each benchmark, less than $1,000$ runs were needed for each program, in line with previous experience [Cucu-Grosjean *et al.* (2012)]. Running $1,000$ times a program whose typical execution time is in the order of few milliseconds (typical value for many CRTES) implies that pWCET estimates for that program can be obtained in a few seconds altogether, which is a rather affordable overhead for an industrial development timescale. Under the heading 'Statistical tests' Table 9.1 reports the results of both tests for all benchmarks. Both tests are passed in all

Table 4.1: Independence and identical distribution test results (2nd and 3rd columns), and average execution time and pWCET bounds of the complex MBPTA-friendly processor vs. an equivalent conventional processor (4th and 5th columns)

Benchmarks	Statistical tests		Timing analysis results	
	Inde-pence	Identical distribution	Average Exec. Time	pWCET 10^{-16}
a2time	0.949	0.387	1.04	1.13
aifirf	0.380	0.791	1.09	1.12
cacheb	0.063	0.668	1.14	1.45
canrdr	0.126	0.529	1.01	1.04
puwmod	0.253	0.993	1.00	1.01
rspeed	0.635	0.628	1.00	1.04
tblook	0.127	0.252	0.87	0.94
ttsprk	0.696	0.187	1.13	1.25

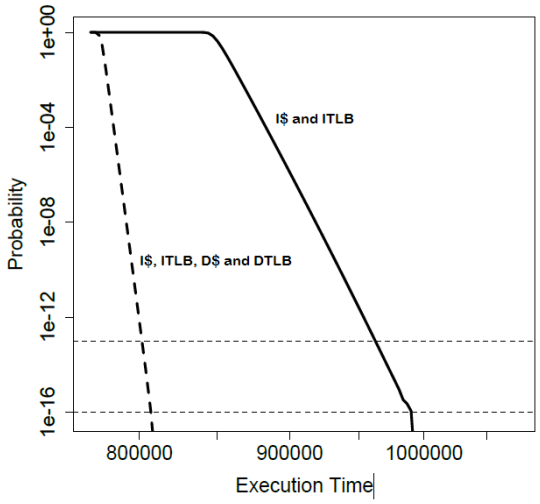


Figure 4.4: pWCET estimates for the *puwmod* benchmark program on different architectural setups.

cases, which proves that the example architecture meets the i.i.d. requirement of our MBPTA approach.

4.4.4 pWCET

In this section we show the type of probabilistic WCET estimates that can be obtained, for the example architecture, with the method presented in [Cucu-Grosjean *et al.* (2012)]. The dashed line in Figure 4.4 plots the pWCET estimates obtained

for the *puwmod* benchmark program of the EEMBC suite, run on the example architecture. The solid line plots the result from running the same program on the example architecture *without* the DC and the DTLB. The X-axis shows the execution time and the Y-axis the probability of exceeding it. Assuming for the sake of argument that an exceedance event represents a timing failure in the system, we may concentrate on exceedance probabilities of 10^{-13} and 10^{-16} per run, in the range of acceptable probabilities of hardware failures in the safety regulation of automotive and avionics systems.

Decreasing the exceedance probability (moving down in the Y axis) does not translate into large increases in the WCET estimates (moving right in the X axis); quite the contrary in fact, as the pWCET curves appear to have a very steep slope. In general, the larger the number of random events entailed in a run (e.g. the number of cache accesses), the less likely abrupt performance variations occur. Thus, execution time variation is moderate and the pWCET curve is steep.

As the example processor architecture demonstrably meets the requirements needed for MBPTA, it can be argued that MBPTA can be applied to performance-aggressive hardware features. To sustain this claim, we observe that the dotted line in Figure 4.4 plots the pWCET estimates obtained after reinstating the DC and the DTLB in the processor architecture. Interestingly, the MBPTA process stays unchanged in procedure and effort, while the pWCET estimates become considerably smaller (around 20% in the specific experiment) owing to the performance boost of using the DC and the DTLB.

To the best of our knowledge, complex architectures including caches, TLB, and staged pipelines with buffers, have not been unrestrictedly used with static timing analysis, unless with cautionary restrictions that mitigate the rapid degradation in the tightness of the WCET estimates that arise from resources being used whose state cannot be determined exactly. Measurement-based timing analysis also is at a loss with those processor architectures, because no suite of observation runs can possibly cover the whole state space of all resources exhaustively. Those techniques are also known to be fragile even to the way the program is built, because small changes in the way program objects are allocated in memory, which are hard to capture in test suites, may lead to abrupt changes in execution time.

4.4.5 MBPTA-friendly Architectures Performance

The above results show that the proposed MBPTA techniques, united with the proposed modification to the design of processor resources, enable CRTES designers to aim at considerably higher levels of guaranteed performance.

Notably, there is a further angle of interest to quantify the benefit of the MBPTA approach discussed in this chapter. Under the heading 'Timing analysis results', Table 9.1 reports average execution times and pWCET estimates

for an exceedance threshold of 10^{-16} per run, obtained for the EEMBC benchmark programs on the example processor architecture. The values are normalised against those obtained running the same programs on an analogous architecture that implements modulo placement LRU replacement caches and TLB instead of random placement and replacement. The average execution time of the MBPTA-friendly architecture is only 4% worse than the time-deterministic alternative, thus showing that time randomisation fares well even in the average case. Even more interesting, pWCET estimates are, on average, only 12% higher than the average performance obtained for a processor architecture implementing LRU as the replacement policy for all caches. Whereas the WCET values for those programs on the time-deterministic architecture are not available (computing them would require the port of static timing analysis tools, which was outside of the scope of this work), relevant literature shows that WCET values are intrinsically very conservative and can be many times greater than the average case [Gustafsson & Ermedahl (2007)]. Our study shows that, for our MBPTA-friendly processor architecture, the observed inflation was up to 45% for `cacheb`, which allows arguing that MBPTA-friendly processors are viable for CRTES industry.

In concluding our outline of the application of MBPTA to our example processor, it is worth noting that the ETPs for individual dynamic instructions in our processor are non-independent across them [Abella *et al.* (2013)]. This is so because random-placement caches (as an instance of a state-sensitive time-randomised resource) create dependence across instructions in the same run since any (random) cache set conflict occurring during a particular run holds systematically across the whole run. Such dependence across ETPs for different instructions is captured in the observations taken at analysis time and hence, is accounted for in the pWCET estimate derived with MBPTA.

4.5 External Results

In the previous section we presented simulation results. As part of the PROXIMA project, it was validated the feasibility and the effectiveness of the proposed MBPTA design principles in a real processor design [Kosmidis *et al.* (2016b)]. In particular, as a result of the collaboration between BSC and Cobham Gaisler, an MBPTA-compliant multi-core processor based on Gaisler's LEON3 processor (used by the European Space Agency in several missions) was designed in RTL level and synthesised on an Altera Stratix IV GX EP4SGX230 FPGA achieving a clock rate of 100 MHz.

The design incorporates both design changes for jittery resources proposed in this Chapter: probabilistic upper-bounding, in the form of time-randomisation, was applied in the cache structures (L1 instruction and data caches, instruction

and data TLBs) and in the bus interconnect between cores, whereas deterministic upper-bounding has been implemented in the floating point unit design.

The time-randomised components are supplied with random numbers by a hardware implemented pseudo-random generator [Agirre *et al.* (2015)]. Cache structures implement both random placement and random replacement as briefly introduced in this Chapter and they are examined in greater detail in the following Chapter. The modifications in the floating point unit are focused on the implementation of the division and square root instructions whose latency is input dependent in the original LEON3 implementation. The division latency varies between 15 and 18 cycles, while the square root between 23 and 26.

The repetition of the experiments described in the previous section on the hardware implementation confirm the same conclusions. The i.i.d. hypothesis is confirmed in all benchmark executions. Also the performance compared to the original non-MBPTA compatible design has been shown to be in the same range with the simulated one. In particular, the average performance is roughly the same with the original design, while the pWCET estimates are on average 9% higher than the average performance of the baseline. Finally, the inflation produced by MBPTA over the execution time measurements has been significantly lower than the ones observed in Section 4.4, up to 8.44%, which is well below the current industrial practice of 20% margin used in the avionics domain [Wartel *et al.* (2013)].

The processor design is included in Gaisler's processors portfolio and it is available for licensing similarly to the rest of Gaisler's IP designs [Gaisler (2016)], thus becoming the first processor in the market compatible with MBPTA.

4.6 Conclusion

In this Chapter we have shown that, in order for the measurement-based variant of probabilistic timing analysis (MBPTA) to be usable economically and assuredly, the target processors should be designed such that every program instruction has a distinct probabilistic Execution Time Profile (ETP). We have shown that this ETP can be built incrementally from the timing behaviour of the processor resources used by that instruction.

Using MBPTA on MBPTA-friendly processor architectures, the timing interference between competing applications, which is one of the key problems in mixed-criticality systems, can be studied from the angle of exceedance probability: the probability that the execution time of a program exceeds a given threshold. We have shown that this threshold is tight, owing to the natural attenuation of multiple worst-case events generated as i.i.d. random variables. We have shown that the probabilistic worst-case execution time bounds obtained with the proposed technique are only marginally greater (around 12% in our case study) than the

average-case performance of time-deterministic processor architectures. This allows achieving higher guaranteed (feasible) utilisation for mixed-criticality systems, because little would be lost, if at all, in raw processor performance, and a great reduction would be had in the pessimistic over-provisioning incurred with traditional techniques. The use of Extreme Value Theory allows setting bounds for execution-time budgets at levels of exceedance probability that satisfy the system assurance requirements. Normal mitigation measures can be taken if protection guarantees had to be provided for higher-criticality applications at conditions past the given exceedance threshold.

Chapter 5

Single Level Hardware Time-Randomised Caches

5.1 Introduction

In the previous chapter we identified the requirements and properties that MBPTA imposes on the hardware we design. As we have discussed, those properties are not met by current processors due to their deterministic nature. For instance, if we run a given program fed with the same input data several times on the same processor, we will observe some variation in its execution time due to, for instance, the fact that it is allocated in different locations in memory resulting in different execution times. However, those execution times are not necessarily probabilistically modellable, i.e. cannot be modelled with random i.i.d. variables.

In this chapter and in the following ones, we focus on the biggest contributor in average and worst case performance, the cache. At the cache level, the deterministic behaviour of the placement (e.g. modulo) and replacement (e.g. least recently used or LRU for short) policies make memory operations not to be modellable probabilistically. According to the design guidelines we introduced in Chapter 4, MBPTA requirements in jittery resources can be satisfied either by redesigning them to work on their highest latency, or by randomising their timing behaviour. Moreover, since the difference between the two possible latencies, hit and miss, usually differ one or more orders of magnitude, the latter option is the most appropriate one.

This solution has already been applied in previous works in the PTA literature and in particular in the form of fully-associative (FA) caches deploying random replacement (RR) policy for both PTA variants, SPTA [Cazorla *et al.* (2013a)] and MBPTA [Cucu-Grosjean *et al.* (2012)]. FA-RR caches allow obtaining an actual hit/miss probability for each memory access, so that when a program runs several times on a processor with such a cache the obtained execution times are modellable

with i.i.d. random variables. Furthermore, the execution time of programs does not depend on their memory layout, thus following the same probabilistic behaviour at both analysis and operation as also required by MBPTA. Unfortunately, only small FA caches can be used in general due to their power hungry and costly implementation, thus constraining PTA applicability.

In this chapter, we focus on the MBPTA variant of PTA. We propose a new random placement policy that allows applying MBPTA not only to FA-RR caches but to set-associative and direct-mapped caches. While random replacement has been used in the past [Cobham Gaisler (2011)] [ARM (2006)], existing placement functions have a purely deterministic behaviour and thus, they cannot be used in the context of MBPTA because there is no way to determine the probability of each cache placement to occur during operation. To solve this problem, we propose a new cache design implementing a *parametric random placement* function with the following properties:

1. The placement function is deterministic during the execution of the program enabling cache lookup to be performed analogously to deterministic-placement caches.

2. Placement is randomised across executions by modifying the seed of the parametric hash function used for placement. In this way, each memory access has hit/miss probabilities so that execution times attain i.i.d. properties as needed for MBPTA.

3. Our cache design has similar average performance to that of deterministic caches, which is important not to jeopardise other metrics such as energy consumption.

4. Similarly to FA-RR caches, our design also reduces drastically the amount of information required by the analysis to derive WCET estimates since knowing the memory addresses accessed is not needed anymore.

5.2 Timing Behaviour of Random Caches

In this chapter we demonstrate that randomising the replacement and placement policies allows constructing ETPs for memory instructions: $(\vec{l}, \vec{p}) = \{l_{hit}, l_{miss}\}\{p_{hit}, p_{miss}\}$, where l_{hit} and l_{miss} are the latency of hit and miss respectively and p_{hit} and p_{miss} the associated probability in each case. In particular, in this section, we show that p_{hit} and p_{miss} can be computed analytically based on the properties of RR and our random placement (RP) policy. As pointed out in the Background, the existence of the ETPs ensures that the execution times are probabilistic and therefore the system fulfils the i.i.d. properties.

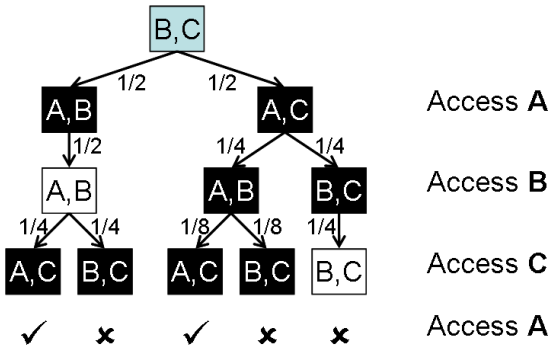


Figure 5.1: Probability tree of the sequence $< A, B, C, A >$. Each box represents the cache state after a given access. Black boxes indicate that the current access misses in cache, while white boxes indicate that the current access hits.

5.2.1 Random Replacement (RR)

RR policy ensures that every time a memory request misses in cache, a way in the corresponding cache set is randomly selected and evicted to make room for the new cache line. This ensures that (1) there is independence across evictions and (2) the probability of a cache line to be evicted is the same across evictions, i.e. for a W-way associative cache, the probability for any particular cache line to be evicted is $\frac{1}{W}$ for each set. In the particular case of a fully-associative (FA) cache, such probability holds for the only cache set.

Given a sequence of cache accesses, the ETP for each of them (i.e. its hit/miss probabilities) can be determined by computing how likely previous accesses can evict the corresponding cache line. For instance, in the sequence $<A,\ B,\ C,\ A>$, B and C can evict A with a given probability that depends on the number of cache ways and whether B and C were fetched before or not. In order to illustrate this, let us assume a FA cache with two ways $W = 2$ that initially contains B and C, each one in a different way. Figure 5.1 shows all possible cache states with their associated probabilities after executing each access in the sequence $<A,\ B,\ C,\ A>$. Black boxes represent cache states in which a miss occurs, while white boxes represent cache states in which a hit occurs. For instance, if the first access to A evicts C (leftmost branch in the tree), B survives and the access B hits in cache. In that case, the next access to C will miss in cache and may or may not evict A (a similar reasoning can be followed for the other subtrees). Overall, the second occurrence of A will hit in cache if and only if the replacement policy does not evict A when B and C are accessed. This means that the second occurrence of A has a hit probability of $\frac{3}{8}$.

Computing hit probabilities for an arbitrarily large number of cache accesses would be costly with this method. For this reason we focus on MBPTA, for which the existence of those probabilities [Cucu-Grosjean *et al.* (2012)] is enough, but there is no need for computing them.

Since cache lines evicted are chosen randomly, whether an access hits or misses *depends solely on random events* for a given sequence of accesses regardless of their absolute addresses, and thus hit/miss outcome is truly probabilistic. In particular, the hit probability (P_{hit}) of a given access A_j in the sequence $< A_i, B_{i+1}, ..., B_{j-1}, A_j >$, where A_i and A_j correspond to accesses to the same cache line and no B_k accesses cache line A, can be approximated as follows, with $P_{miss} = 1 - P_{hit}$ for any access:

$$P_{hit_{A_j}} = \left(\frac{W-1}{W}\right)^{\sum_{k=i+1}^{j-1} P_{miss_{B_k}}} \tag{5.1}$$

$P_{hit_{A_j}}$ is the probability of A surviving all evictions performed by $< B_{i+1}, ..., B_{j-1} >$, which depends on the probability of surviving one eviction times the number of evictions in that sequence. The probability of A to survive one random eviction is $\frac{W-1}{W}$. Meanwhile, given that one random eviction is performed on every miss, the total number of evictions equals the expected number of misses in between A_i and A_j, which is $\sum_{k=i+1}^{j-1} P_{miss_{B_k}}$. In the worst case $P_{miss_{B_k}} = 1$ and so $\sum_{k=i+1}^{j-1} P_{miss_{B_k}} = (j - i - 1)$.

Using Equation 5.1 the hit/miss probabilities of each access can be approximated sequentially starting from the first cache access. If no access has been performed to A before A_j, then the hit probability is zero. Overall, the use of RR allows deriving an ETP for each memory operation, thus enabling MBPTA.

Cache layouts: In this subsection, we further elaborate on the distinct number of cache states during the execution of a sequence of accesses, and introduce the concept of *cache layout* as a means to link random replacement and random placement caches.

On every access to a FA cache, an associative search is done among all the different cache ways. On an eviction, the new fetched cache line can be placed in any cache way. We define *cache layout* as the resultant address-to-cache-line mapping after assigning one or several memory objects into the cache. In the case of a FA cache, a new *cache layout* is built at every cache miss because, on every miss the newly fetched address (object) is placed in a random line.

Table 5.1: Cache layouts for the sequence $<A,\ B,\ C,\ A>$ in a 2-way FA cache

Way 1	**Way 2**
A_1, B, C, A_2	
A_1, B, C	A_2
A_1, B, A_2	C
A_1, C, A_2	B
B, C, A_2	A_1
A_1, B	C, A_2
A_1, C	B, A_2
A_1, A_2	B, C

This makes, for a given sequence of memory addresses, the number of cache layouts depend on the number of evictions happening in that sequence, which is bounded by the number of cache accesses in the sequence (m). For instance, $m = 4$ in our previous example ($<A,\ B,\ C,\ A>$). The total number of cache layouts is thus upper-bounded by the m^{th} *moment of a probability distribution of random permutations* [Graham *et al.* (1988)]:

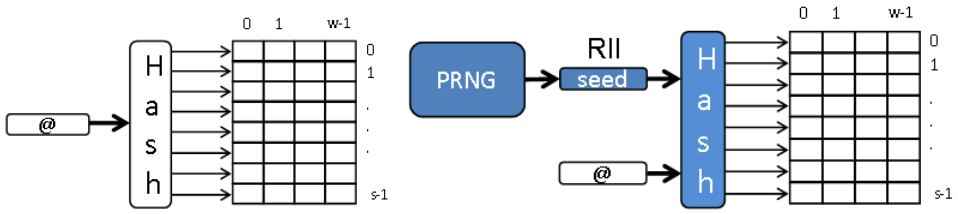
$$E(X^m) = \sum_{j=0}^{W} S_{m,j} \tag{5.2}$$

where X stands for the random variable that models the cache behaviour, i.e. the random replacement policy, and $S_{m,j}$ stands for the Stirling number of the second kind with parameters m and j [Cargal (1988)]. The m^{th} moment is the number of partitions of a set of cache accesses (m) into no more than W cache ways. In our example, $E(X^m) = 8$. Table 5.1 provides the 8 different cache layouts for our example, where A_1 and A_2 stand for the first and second occurrence of A respectively. Note that the order is not relevant for cache layouts since, for instance, mapping A_1, B and C in the first cache line and A_2 in the second line is analogous to mapping A_2 in the first cache line and A_1, B and C in the second line.

5.2.2 Random Placement (RP)

The RP policy we are after has to ensure that the cache set in which a cache line is mapped is randomly selected. Ideally, assuming a cache with S sets, the probability for a cache set to be selected is $\frac{1}{S}$.

One fundamental difference between placement and replacement policies is that placement assigns sets to cache lines based on the index bits of the memory address,



(a) standard (b) proposed

Figure 5.2: Block diagram of the cache design.

Figure 5.2(a). As a result, if the placement policy assigns two memory addresses to the same cache set, they will collide systematically during the entire execution of the program. To deal with this deterministic nature while randomising the timing behaviour of the placement policy, we propose a new parametric hash function that makes use of a random number as an input. Such random number can be generated either by hardware or software. Our hash function, given a memory address and a random number called *random index identifier* (RII), provides a unique and constant cache set (mapping) for the address along the execution, see Figure 5.2(b). If the RII changes, the cache set in which the address is mapped changes as well, so cache contents must be flushed for consistency purposes. We propose changing the RII only across program execution boundaries so that programs can be analysed with end-to-end runs without any further consideration than assuming that the cache is initially empty[1]. We assume that given a memory address and a set of RIIs, the probability of mapping such address to a given cache set is the same, i.e. $\frac{1}{S}$, although this is not needed as long as such mapping is probabilistic. How we approximate this ideal distribution by hardware is shown in Section 5.3.

Next we describe how to quantify the probability of each memory address to be mapped into a given cache set, and so conflicting with other memory addresses. Given u different memory objects and S cache sets, a new *cache layout* results when the placement policy assigns (maps) the u memory objects into the S cache sets. Every time the program is executed, a new RII is generated leading to a new random mapping function corresponding to a new *cache layout*.

Note that different cache layouts cause different cache conflicts among memory addresses, resulting in different execution times. Further note that different cache layouts may lead to the same execution time. For instance, if we have three memory

[1]The RII could also be changed periodically as a means to increase the degree of randomisation; however, it would be necessary to flush caches. The maximum execution time impact of this process (cache flushing and serving as many misses as cache lines) should be then accounted for. As shown in [Wartel *et al.* (2013)], this is not needed for real applications as the degree of randomisation achieved by changing the RII across execution boundaries is enough to obtain WCET estimates close to the actual performance on conventional caches.

Table 5.2: Possible cache layouts for the different accesses of the sequence <*A, B, C, A*> in a idealised random placement cache with two sets.

Cache layout	Conflicts (id)	P_{layout}
$A_0B_0C_0$	*A, B* and *C* (1)	$(\frac{1}{2})^3$
$A_0B_0C_1$	*A* and *B* (2)	$(\frac{1}{2})^2(1-\frac{1}{2})$
$A_0B_1C_0$	*A* and *C* (3)	$(\frac{1}{2})^2(1-\frac{1}{2})$
$A_0B_1C_1$	*B* and *C* (4)	$(1-\frac{1}{2})^3$
$A_1B_0C_0$	*B* and *C* (4)	$(1-\frac{1}{2})^3$
$A_1B_0C_1$	*A* and *C* (3)	$(\frac{1}{2})^2(1-\frac{1}{2})$
$A_1B_1C_0$	*A* and *B* (2)	$(\frac{1}{2})^2(1-\frac{1}{2})$
$A_1B_1C_1$	*A, B* and *C* (1)	$(\frac{1}{2})^3$

objects (*A*, *B* and *C*) and 4 cache sets, any cache layout where *A* is mapped in one cache set (e.g., set 0) and *B* and *C* in a different cache set (e.g., set 1) will be equivalent in terms of execution time.

To develop our argument let us assume a random placement direct-mapped cache composed of $S = 2$ cache sets, where no replacement policy is needed and hence, only the placement determines the cache layout and so the execution time. Table 5.2 identifies all possible cache layouts of a program consisting of $u = 3$ memory objects mapping into different cache lines (<*A, B, C, A*>). The subscript in each address in the first column indicates the cache set on which each address is mapped, 0 or 1 in this example.

With random placement we can derive the probability of each cache layout to occur. The column labelled as P_{layout} in Table 5.2 shows the probability of each cache layout to happen: The probability of the cache layout in which *A, B* and *C* are mapped into the same set ($A_0B_0C_0$ and $A_1B_1C_1$) is $(\frac{1}{2})^3$ each. Similarly, the probability of the cache layout in which *A* is mapped in a different entry to *B* and *C* ($A_0B_1C_1$ and $A_1B_0C_0$) is $(1-\frac{1}{2})^3$.

However, we are not interested in all cache layouts, but only in those that may produce different execution times. For instance, in Table 5.2, cache layouts $A_0B_0C_1$ and $A_1B_1C_0$ result in exactly the same cache conflicts (and so the same execution time), because they will experience exactly the same misses under both cache layouts. The total number of cache conflict layouts is given by the u^{th} *moment of a probability distribution of random permutations* [Graham *et al.* (1988)]:

$$E(X^u) = \sum_{j=0}^{S} S_{u,j} \tag{5.3}$$

The u^{th} moment is the number of partitions of u unique memory addresses into no more than S cache sets. Thus, $E(X^u)$ provides the number of unique cache

conflicts among the u memory addresses. In the example above, the number of possible cache layouts is 4, identified by a number in parenthesis in the second column of Table 5.2.

With this, we can compute the probability of each cache layout and hence the probability of its resulting execution time for a given program. If we consider the example shown in Table 5.2, and we assume a hit latency of 1 cycle and a miss latency of 10 cycles, we can derive the following probability distribution function for the observed execution times: $\{(31, 40), (0.25, 0.75)\}$. Cache layouts (1), (2) and (3) lead to an execution time of 40 cycles with an associated probability of 0.25 each. Cache layout (4) leads to an execution time of 31 cycles with an associated probability of 0.25. Note that still non-equivalent cache layouts may lead to the same execution time as it is the case for cache layouts (1), (2) and (3) depending on how accesses to the u memory objects interleave. Thus, analogously to $E(X^m)$ for random-replacement caches, $E(X^u)$ is an upper bound of the number of cache layouts for random-placement caches.

By using a new RII on each execution, a random cache layout is chosen and pathological scenarios can only occur with a given probability. This allows deriving a hit probability for each access.

In an arbitrary sequence $A, B_1, B_2, ...B_q, A$ where $\forall i, j : i \neq j$ and $B_i \neq B_j$, the probability of the second occurrence of A to survive (and so being a hit) in a direct-mapped cache is determined by those cache layouts in which the q objects in between are placed in a different cache set to A. If we consider that A is placed in a particular set, the number of cache layouts in which the other q objects are placed in different cache sets is $(S-1)^q$: the q objects can be placed in all sets except the one where A is placed. Because A can be placed in any position, the number of cache layouts in which A survives is $(S-1)^q \cdot S$. Therefore, and considering that the number of possible cache layouts is S^{q+1}, the probability of the second occurrence of A being a hit can be approximated using the following equation:

$$P_{hit_A} = \frac{(S-1)^q \cdot S}{S^{q+1}} = \left(\frac{S-1}{S}\right)^q \tag{5.4}$$

The reuse distance of A, defined as the number of unique cache line addresses (q) between two occurrences of the same memory address, determines how likely A will result in a hit/miss. The higher the q-distance is between two occurrences of A, the less likely the second occurrence of A to survive. For instance, A is more likely to be evicted in the sequence *A, B, C, A* ($q = 2$) than in the sequence *A, B, B, B, B, B, A* ($q = 1$).

Overall, the hit probability for any access exists (and so the miss probability). Therefore, the use of RP allows deriving an ETP for each memory operation, thus enabling MBPTA as it is the case for RR, because execution times will be i.i.d.

5.2.3 Putting All Together: Set-Associative Caches

The number of cache layouts for a set-associative cache implementing random placement and replacement can be computed as the combination of the number of cache layouts provided by the placement and replacement policies. Thus, the number of cache conflict layouts can be computed as the product of the u^{th} and m^{th} moments of a probability distribution of random permutations:

$$E(X^m) \cdot E(X^u) = \sum_{j=0}^{W} S_{m,j} \cdot \sum_{j=0}^{S} S_{u,j} \tag{5.5}$$

$E(X^u)$ and $E(X^m)$ are the number of cache layouts given by the random placement and random replacement policies respectively.

For example, if we consider a program composed of the sequence of memory accesses <*A, B, C, D, A, B, A, C, A, D, A, B, A, C, A, D*>, in which $u = 4$ and $m = 16$, and a cache with 4 cache lines, we can compute the number of cache layouts for different cache configurations. In particular, for this example, we consider a direct-mapped random-placement (DM-RP) cache with $S = 4$, a set-associative random placement and random replacement (SA-RP+RR) cache with $S = 2$ and $W = 2$, and a fully-associative random replacement (FA-RR) cache with $W = 4$.

The number of cache layouts of those cache configurations is 15 for DM-RP, 26,244 for SA-RP+RR and 178,973,355 for FA-RR. It can be seen that the higher the associativity, the higher the number of cache layouts is. This is an expected result because the number of cache layouts for DM and FA caches depends on the number of unique cache line addresses (u) and the number of total cache accesses (m) respectively, and $u \leq m$. The number of cache layouts for SA caches must be always somewhere in the middle. This has an important implication in the worst-case performance of caches. Both caches have the same worst-case cache layout, i.e. the one in which all memory objects are mapped into the same cache entry, resulting in the largest number of cache misses. However, the probability of experiencing such cache layout is much lower for the FA-RR cache. In our example such probability is 1/15 and 1/178973355 for the DM-RP and FA-RR caches respectively. Hence, by using a set-associative cache with random placement and replacement policies (SA-RP+RR) the probability of experiencing the worst-case performance decreases rapidly with respect to the DM-RP: the number of cache layouts increases as the degree of randomness increases as well.

The ETP of a memory operation accessing to a $S \cdot W$ set-associative cache with random placement and replacement policies is the combination of the ETPs of both policies. That is, the random placement will allocate memory objects into

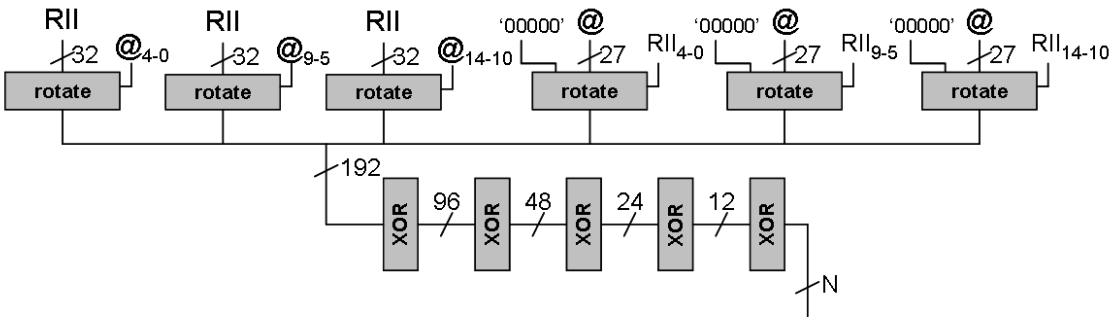


Figure 5.3: Parametric hash function proposed for the random-placement cache.

the S sets with a probability of $\frac{1}{S}$ while the random replacement policy will evict a way to allocate a new fetched cache line with a probability of $\frac{1}{W}$. In particular, given the sequence $< A_i, B_1, ..., B_k, A_j >$, where A_i and A_j are two accesses to the same cache line and no B_l (where $1 \leq l \leq k$) accesses cache line A_i the probability of miss of A_j can be formulated as follows:

$$P_{miss_{Aj}}(SA[S,W]) = P_{miss_{Aj}}(DM[S]) \cdot P_{miss_{Aj}}(FA[W]) \tag{5.6}$$

By combining Equation 5.1 and Equation 5.4 into Equation 5.6, the probability of miss and hit respectively of A_j can be approximated as:

$$P_{miss_{A_j}}(S,W) = \left(1 - \left(\frac{W-1}{W}\right)^{\sum_{l=1}^{l=k} P_{miss_{B_l}}}\right) \cdot \left(1 - \left(\frac{S-1}{S}\right)^{k}\right) \tag{5.7}$$

$$P_{hit_{A_j}} = \left(\frac{W-1}{W}\right)^{\sum_{l=i+1}^{j-1} P_{miss_{B_l}}} \cdot \left(\frac{S-1}{S}\right)^{k} \tag{5.8}$$

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

$$\begin{aligned} ETP_{cache} = \{\!\{&l_{hit}, l_{miss}\}, \\ &\{P_{hit_{A_j}}(S,W), 1 - P_{hit_{A_j}}(S,W)\}\} \end{aligned} \tag{5.9}$$

In summary, hit/miss probabilities exist for all accesses, and so their ETPs. As a consequence, execution times will be i.i.d. and MBPTA can be safely applied on top of a SA cache.

5.3 Hardware Design of a Random Cache

This section describes how to implement both random placement and replacement policies.

5.3.1 Random Replacement

Random replacement policies have been extensively used in various processor architectures, both in the high-performance and embedded markets. Examples for the latter market are the Cobham Gaisler NGMP [Cobham Gaisler (2011)] or some processors of the ARM family [ARM (2006)]. The most relevant element of a random replacement policy is the hardware generating random numbers which selects the way to be evicted on a miss. In general, pseudo-random number generators (PRNG) are implemented. Given that efficient implementations of a PRNG exist in the aforementioned processors, we omit the details of our implementation of the PRNG. The particular PRNG we have used in our proposal is the Multiply-With-Carry (MWC) [Marsaglia & Zaman (1991)] PRNG, since we have tested that (i) it generates numbers with a sufficiently high level or randomness, (ii) its period is huge, and (iii) it can be efficiently implemented in hardware. Further work on PRNGs in the context of MBPTA has been done in [Agirre *et al.* (2015)] proving that they can be used in high-integrity systems and go through strict certification processes.

5.3.2 Random Placement

In this section, we propose an implementation of a random placement policy. The key components of this design are (1) a low-cost PRNG if the RII is produced by hardware and (2) a *parametric hash function*, see Figure 5.2(b).

In order to keep cache latency and energy low, the implementation of both components must be kept simple. Moreover, both components are placed 'in front' of the cache, so the cache design is not changed, see Figure 5.2(b), but some extra logic is added before accessing cache. As for random replacement, we use the MWC PRNG if the RII is produced by hardware.

The Parametric Hash Function is used to randomise cache placement. To be effective, the parametric hash function must have the following two properties:

- Small variations in the address bits or the RII must produce arbitrary variations in the output bits determining the set where the address is mapped.
- For any given pair of addresses, modifying the RII must produce different variations so that both addresses do not collide systematically in the same

cache set. Ideally, both addresses should collide into the same set with a probability around $\frac{1}{S}$ for different RII values.

Those properties can be attained if the address and RII bits are used to control rotating blocks whose input is, for instance, the address to be accessed or the RII. In particular, RII bits must be used to control a rotate block[1] whose input is the address or vice versa. A single bit modification in either the address or the RII can change all output bits of the rotate block if such bit is one of the control bits. Note that using address bits to control a rotate block whose input is the address itself would be ineffective because changing the RII across runs would have no effect. Analogously using the RII bits only would also be ineffective because variations in the output of the rotate block would be identical for all addresses.

Figure 5.3 shows our implementation of the parametric placement function. The hash function has two inputs, the bits of the address used to access the set (index bits), '@' in Figure 5.3, and a RII. In the configuration of the particular example, 32 bytes per cache line and 32-bit addresses are assumed. Therefore, the 5 lowermost bits are discarded (offset bits) and only 27 bits are used.

The hash function rotates the address bits, based on some bits of the RII as it is shown in the three rightmost rotate blocks of the figure. By doing this, we ensure that when a different RII is used, the mapping of that address changes, and this change is different for different addresses. Analogously, the RII bits are rotated based on some address bits to obtain a different layout of RII bits for each address. This operation, which is performed by the three leftmost rotate blocks, changes the way the address bits are shifted. Note that addresses are padded with zeros to obtain a power-of-two number of bits, so address bits can be rotated without any constraint. Otherwise, rotation values between 27 and 31 would require special treatment.

Finally, all bits of the rotated addresses, the original address and the RII (192 bits in the example), are XORed successively, until we obtain the desired number of bits for indexing the cache sets. For example, a 16KB cache with 32 bytes per line would need 9 index bits for a direct-mapped organisation, 8 bits for a 2-way set-associative, and so on and so forth. Hence, 5 XOR gate levels are enough to produce the index. The number of rotate blocks in each group is odd deliberately to avoid the case where bits controlling the rotation match across rotate blocks, thus producing the same output bits, which could systematically generate zeros when XORed.

As shown in Figure 5.3, the hardware implementation of the hash function consists of 6 rotate blocks and 5 levels of 2-input XOR gates. Each rotate block can be implemented with a 5-level multiplexer [Huntzicker *et al.* (2008)]. Since

[1]Control bits of a rotate block are those determining how many positions the input bits are rotated.

the latency and the energy per access of a fully-associative cache is much larger than the one of direct-mapped or set-associative caches, the relative overhead of the hash function is small. We have corroborated this observation by integrating our parametric hash function into the CACTI tool [Muralimanohar *et al.* (2009)]. Results for several cache configurations show that energy per access grows less than 4% and delay grows by 40% (it is still less than half the delay of a fully-associative cache). Note that hit latency has low impact in WCET since it is typically some orders of magnitude lower than miss latency. Nevertheless, we assume the same hit latency for our DM and SA configurations, and the FA one, which plays against our proposal. Detailed power and delay results are provided later in Section 5.4.7.

5.4 Results

5.4.1 Experimental Setup

We use the experimental setup (simulation framework and EEMBC benchmarks) described in Chapter 3. Both instruction and data cache size is 4-KB with 16-byte line size. Associativities considered are 1-way (direct-mapped); 2-way, 4-way, 8-way and 32-way (set-associative); and 256-way (fully-associative). Both caches implement random replacement and our random placement policy. We assume 100-cycles memory access latency.

The latency of the fetch stage depends on whether the access hits or misses in the instruction cache: 1 cycle in case of hit and 100 in case of miss. After the decode stage, memory operations access the data cache so they can last 1 or 100 cycles depending on whether they miss or not. The remaining operations have a fixed execution latency (e.g. integer additions take 1 cycle).

We model a single-core processor. WCET estimates are obtained for run-to-completion executions. Studying the interaction between our time-randomised cache design in the presence of system effects such as preemptions is out of the scope of this thesis and is part of our future work. Our results in Chapter 11 show, that time-randomised caches simplify time composability, one of the most important metrics to optimise in current and future integrated real-time systems.

5.4.2 Quality of the Parametric Hash Function Implementation

We have evaluated the quality of our random placement function by using the test battery provided by the US National Institute of Standards and Technology [Rukhin *et al.* (2010)]. Those tests evaluate the quality of the bit sequences produced by the PRNGs by studying the distribution of ones and zeros, their pat-

terns, whether subpatterns repeat, etc. In particular, we have generated a sequence of 40,000,000 bits consisting of the set number produced by our parametric hash function given a particular random address and a sequence of 5,000,000 random numbers. Given that the cache considered is the same as above (4KB direct-mapped 16B/line), there are 256 cache sets and thus, each set identifier consists of 8 bits. Our hardware implementation of the parametric hash function passed 99.9% of the tests. None of the 9 PRNGs provided together with the test battery achieved a higher pass rate. Only two of them obtained the same pass rate, the *Secure Hash Generator* and the *Micali-Schnorr Generator* whose implementation is described in [Rukhin *et al.* (2010)].

In the same experiment we have also tested the distribution across sets that our parametric hash function achieves. Given 5,000,000 set identifiers and 256 sets, we should expect our function to generate around 19,531 times each set identifier. Results show that the normalised standard deviation of the counts for the different sets is only 0.46% with respect to the expected value (90.4 with respect to 19,531). Maximum and minimum counts obtained are 19,769 and 19,333, so 1.2% and 1.0% away from the expected value.

Finally, we have tested how independent is the randomisation of different addresses. For that purpose we have generated 4 random addresses (say A, B, C and D) and have evaluated how many times addresses B, C and D are mapped into the same set as A for 2^{30} random seeds. In the ideal case we should obtain 2^{22} matches for each individual address (e.g., B being mapped into the same set as A), 2^{14} for each different pair of addresses (e.g., B and C being mapped into the same set as A simultaneously) and 2^6 matches for the case where all addresses are mapped simultaneously in the same set. Results show that individual matches are within 0.2% of the ideal value (4,186,839 real vs. 4,194,304 expected), pairs are within 1.3% of the ideal value (16,591 real vs. 16,384 expected) and all of them collide in the same set 59 times when the expected value is 64 times.

Overall, we can conclude that our hardware implementation of the parametric hash function achieves both (i) a high degree of randomisation, (ii) a near-optimal distribution of placement choices across sets and (iii) independent randomisation of addresses.

5.4.3 Behaviour of the Parametric Hash Function Implementation

In order to compare the behaviour of the implementation of our parametric hash function with an idealised random placement where each address is randomly mapped into one of the S sets, we compute the complementary cumulative distribution function (CCDF) of a representative benchmark, *a2time* for both schemes

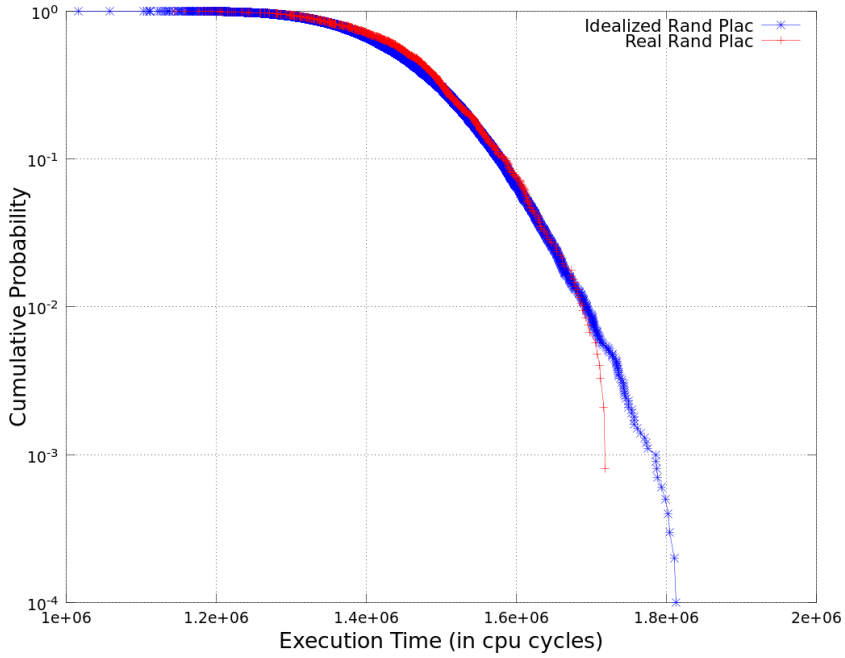


Figure 5.4: 4KB direct-mapped cache considering an idealised random placement and the actual hardware implementation of the random placement (labelled as *Idealised Rand Plac* and *Real Rand Plac* respectively).

(labeled *Real Rand Plac* and *Idealised Rand Plac* respectively) for a 4KB direct-mapped cache as shown in Figure 5.4. It can be seen that there is not meaningful difference between the two CCDF. Results only differ slightly in the tail of the queue. This indicates that the actual implementation of the parametric hash function randomises address mapping quite well across sets. Discrepancies in the tail of the queue can occur due to the loss of accuracy of the measurements for that region[1]. The results obtained for the other benchmarks match those presented in Figure 5.4.

5.4.4 Fulfilling the i.i.d properties

The use of random replacement and placement policies guarantees that an ETP exists by construction for each memory operation, and so observed execution times fulfil the properties required by MBPTA. We further verify this point empirically by analysing whether execution times of EEMBC benchmarks on 6 different cache configurations (see Section 5.4.1) are independent and identically distributed.

[1]Few measurements build the tail of the distribution. The particular behaviour of each of those measurements due to purely random events may create deviations with respect to the real ideal distribution.

Table 5.3: IPC of RP+RR and modulo+LRU caches in our architecture

		1w-256s DM	**2w-128s SA**	**4w-64s SA**	**8w-32s SA**	**32w-8s SA**	**256w-1s FA**
RP+RR	avg	2.10	1.78	1.73	1.73	1.74	1.75
	std	0.32	0.12	0.08	0.06	0.04	0.01
mod+LRU	avg	2.67	1.57	1.60	1.61	1.64	1.74

In order to test independence we use the Wald-Wolfowitz independence test [Bradley (1968)]. We use a 5% significance level (a typical value for this type of tests), which means that absolute values obtained after running this test are lower than 1.96 if there is independence, and higher otherwise. For identical distribution, we use the two-sample Kolmogorov-Smirnov identical distribution test [Boslaugh & Watters (2008)] as described in [Cucu-Grosjean *et al.* (2012)]. For 5% significance, the outcome provided by the test should be above the threshold (0.05) to indicate identical distribution, and non-identical distribution otherwise.

Our results show that output values for the independence test values are largely below the threshold (1.96 for a 5% threshold) and p-values for the KS test are well above 0.05, thus proving that all our cache designs provide i.i.d. as needed by MBPTA.

5.4.5 Performance Analysis

Next, we compare the average performance of deterministic and random caches. In particular, we compare different *random placement+replacement* (RP+RR) caches against *modulo placement and LRU replacement* (mod+LRU) caches for different associativities. Table 5.3 shows the average CPI (cycles per instruction) for all EEMBC benchmarks under different cache configurations and 1,000 runs per benchmark for RP+RR caches. Standard deviation is also shown for RP+RR caches.

- As shown, execution time variation is quite stable for cache in the range 4-way to 256-way. Most of the benchmarks get little benefit due to the extra associativity and, in fact, some of them lose some performance (higher CPI) when increasing the associativity due to capacity misses. Note that in the extreme case, a program with a working set slightly larger than the cache size suffers more misses in a fully-associative cache than in a lowly-associative one. This is so because in lowly-associative caches some data may fit in few particular sets and deliver extra hits in front of a fully-associative one where all data are replaced short before being reused. This is the case for some programs in our setup (for instance, *aifftr* and *aiifft*). This effect

is less noticeable for RP+RR caches because randomness breaks systematic pathological cases.

- When associativity becomes very low (2-way caches), most of the benchmarks observe little performance variation for mod+LRU caches and those exceeding slightly cache capacity get further improvements (so lower CPI). As stated before, this effect is not so relevant for RP+RR caches, where some benchmarks observe some execution time degradation with 2-way caches due to the conflicts introduced by random placement in some runs.

- Finally, if 1-way (direct-mapped) caches are used, conflicts dominate cache behaviour. While this introduces plenty of pathological cases in mod+LRU caches, pathological cases occur only randomly for RP+RR caches. Therefore, RP+RR caches offer lower execution time (so lower CPI) than mod+LRU under direct-mapped setups.

If we consider execution time variations in RP+RR caches, we observe that decreasing cache associativity may lead to more extreme scenarios. For instance, if a program accesses few different cache lines, those can evict each other some times in a fully-associative cache until they are finally placed in different cache lines. If we use a lowly-associative cache (e.g., 2-way or 4-way) most of the times those addresses will be placed into different cache sets, thus leading to slightly higher performance (so lower CPI) than a fully-associative cache. However, in some cases those addresses will be mapped into the same cache set and will experience many more conflicts, thus increasing the execution time (and so the CPI). As a consequence, although the average CPI is roughly the same for RP+RR cache configurations between 2-way and 256-way, their CPI variation increases as associativity decreases. As shown later, this variation has a direct impact on the pWCET estimates obtained for low exceedance probabilities. Just as a matter of fact, if we consider the average execution time plus 3 times the standard deviation ($\mu + 3 \cdot \sigma$), we can expect the execution time of around 99.9% of the runs to be below this value, so we should expect only 1 every 1,000 runs to exceed it. Then, we realise that this value is 1.78 for 256-way caches and 2.14 for 2-way caches. Thus, although higher associativity may not provide better average CPI, it provides more stable performance.

Example: Detailed results for *tblook*, *rspeed* and *aifftr* benchmarks are shown in Figures 5.5, 5.6 and 5.7 respectively. *tblook* shows some significant CPI degradation when associativity decreases down to 2-way and 1-way. Also, variability for RP+RR caches increases as associativity decreases. *rspeed* instead is the example of a highly stable benchmark fitting very well in cache so that its CPI is highly insensitive to cache associativity. Only for direct-mapped RP+RR caches variability increases due to few executions where those few cache lines reused are

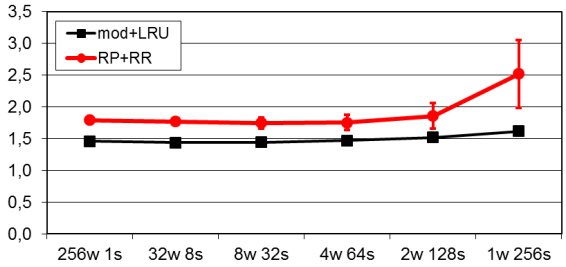


Figure 5.5: CPI (cycles per instruction) for *tblook* benchmark for some RP+RR and LRU+mod cache configurations. In particular, we show, from left to right, FA, 32-way, 8-way, 4-way, 2-way and DM caches.

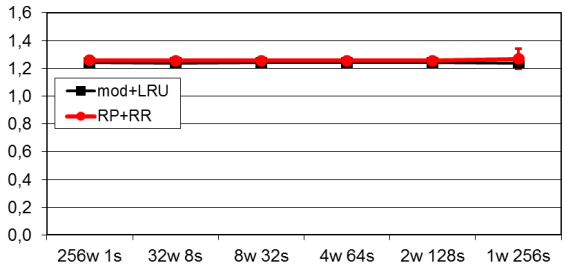


Figure 5.6: CPI (cycles per instruction) for *rspeed* benchmark for some RP+RR and LRU+mod cache configurations. In particular, we show, from left to right, FA, 32-way, 8-way, 4-way, 2-way and DM caches.

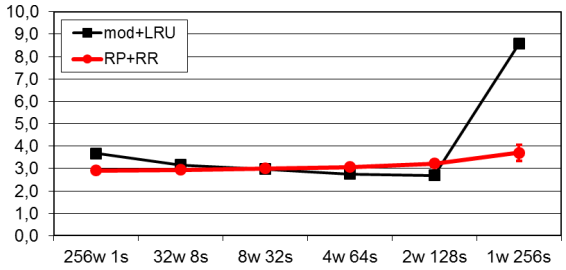


Figure 5.7: CPI (cycles per instruction) for *aifftr* benchmark for some RP+RR and LRU+mod cache configurations. In particular, we show, from left to right, FA, 32-way, 8-way, 4-way, 2-way and DM caches.

randomly mapped into the same cache set. *aifftr* is a clear example of a program not fitting completely in cache. Therefore, its execution time decreases as associativity decreases for mod+LRU configurations, except when a direct-mapped cache is used because then conflicts dominate its behaviour. Instead, RP+RR caches show much more stable performance across different cache setups. Conflicts across different cache lines are abundant in all setups, but extreme conflicts never occur due to the low probability of extreme scenarios. This can be easily explained with

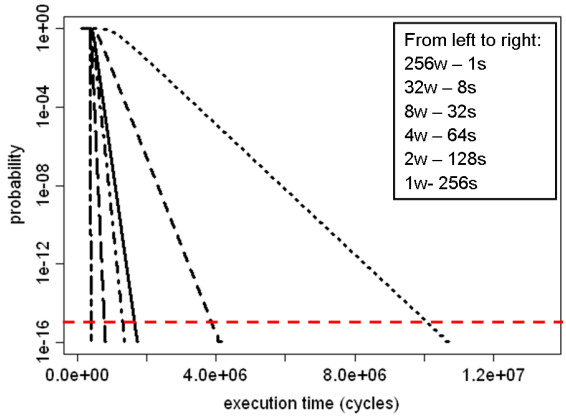


Figure 5.8: EVT projection for *a2time*.

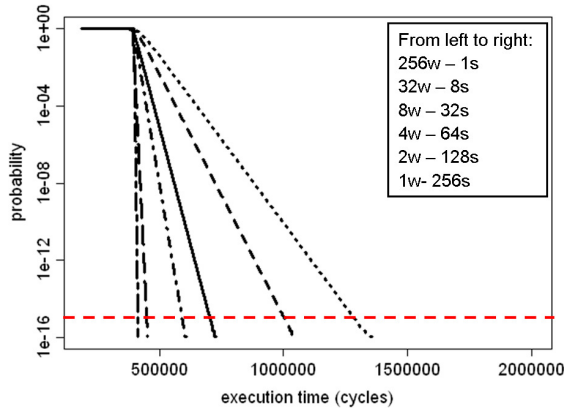


Figure 5.9: EVT projection for *ttsprk*.

an example. If we flip 1,000 coins, it is extremely unlikely to get a number of tails (or faces) out of the range 400-600. In the case of mod+LRU, those 1,000 coins have been glued to each other, so whatever it happens, is either very good (no conflict at all) or very bad (systematic conflicts).

5.4.6 MBPTA: EVT projections

In this section we provide several pWCET estimates obtained with the method provided in [Cucu-Grosjean *et al.* (2012)]. Note that MBPTA has been used so far *only* on top of FA-RR caches. Although FA-RR caches fulfil the properties required by MBPTA, they have high hardware implementation cost and low scalability. Therefore, this chapter provides the first SA and DM cache designs amenable for MBPTA.

Our selection of the exceedance probability, 10^{-15}, i.e. the probability that an instance of a task misses its deadline, is based on the allowed probabilities for random hardware failures in safety standards in the automotive and avionics domains, as in previous chapters.

Following the iterative method described in Chapter 3 we carried out 1,000 experiments and used EVT to extract pWCET estimates. Figures 5.8 and 5.9 show the EVT projections generated with MBPTA [Cucu-Grosjean *et al.* (2012)] for `a2time` and `ttsprk` considering a FA-RR cache (labelled as *256w-1s*), several set-associative RP+RR caches (labelled as *32w-8s*, *8w-32s*, *4w-64s* and *2w-128s*) and a DM-RP cache (labelled as *1w-256s*). As expected, the FA-RR cache provides the lowest pWCET estimates. That is, the random replacement policy has lower probability of resulting in cache layouts with multiple cache conflicts because random choices are taken on every miss instead of across different runs. However, as we reduce the associativity of the cache, and so we increase the number of sets, the number of cache layouts decreases, thus increasing the probability of having more cache conflicts.

The pWCET increment due to the reduction of the cache associativity depends on the application: for instance, *a2time* is very sensitive to cache associativity as shown in Figure 5.8. For an exceedance probability of 10^{-15}, the pWCET of a 8-way cache for *a2time* grows 311% with respect to the FA-RR cache and more than 24x in the case of the DM-RP cache. Instead, *ttsprk* experiences pWCET estimate increments of only 9% and 43% when considering 32-way and 8-way caches and around 3x when considering a DM-RP cache, with respect to the FA-RR cache.

Table 5.4 shows the pWCET increment of all set-associative and direct-mapped caches with respect to the fully-associative one for all benchmarks when considering an exceedance probability of 10^{-15}. Overall, our RP+RR cache designs provide the best tradeoff between hardware complexity and pWCET for MBPTA while not requiring information about the actual addresses accessed by the programs analysed. Moreover, second level caches can be used to mitigate the large impact of misses in both average performance and pWCET estimates. We discuss second level-caches in the next Chapter.

The lower the associativity the higher the pWCET bound is. The reason for that behaviour is that, although average performance does not change noticeably across cache configurations, execution time variation grows when decreasing cache associativity. While this variation may be relatively small in the 1,000 runs of a particular benchmark for a given cache setup, it may grow orders of magnitude at very low probabilities (e.g., 10^{-15}), and so MBPTA must account for that. Thus, pWCET estimates for direct-mapped caches are around 10X those for fully-associative ones even if average execution time is only 20% higher. For reasonable cache setups pWCET increments with respect to the ideal fully-associative cache

Table 5.4: pWCET increment of the SA and DM caches with respect to the FA one, considering an exceedance probability of 10^{-15}

Benchmarks	32w-8s (SA)	8w-32s (SA)	4w-64s (SA)	2w-128s (SA)	1w-256s (DM)
a2time	95%	228%	311%	862%	2404%
aifftr	39%	41%	68%	138%	532%
aifirf	111%	152%	317%	452%	777%
aiifft	38%	53%	54%	97%	571%
cacheb	1%	10%	16%	75%	1008%
canrdr	11%	26%	90%	191%	614%
iirflt	178%	329%	419%	543%	1945%
puwmod	16%	36%	45%	250%	626%
rspeed	9%	22%	84%	230%	475%
tblook	56%	152%	185%	299%	784%
ttsprk	9%	43%	70%	142%	210%

are far more moderate. For instance, they are 99% and 151% higher for 8-way and 4-way caches respectively than for fully-associative caches. Note that, as stated earlier, cache sizes (4KB) have been chosen to create conflict and capacity misses in several benchmarks. Thus, if larger caches are used instead (e.g., 8KB or 16KB) or if L2 caches are set up to mitigate miss latencies, then pWCET estimates for lowly-associative caches would get closer to those of fully-associative ones.

Results have been obtained assuming that cache latency is not increased due to the hash function. If this is not the case, cache latency should be increased by up to 1 cycle. We have corroborated that, as stated before, increased cache latency has negligible impact in pWCET estimates, which grow on average between 1.1% and 5.5% only across different associativities.

5.4.7 Power and Delay Analysis

This section evaluates the power and delay overhead of the parametric hash function when used together with different cache configurations. We have integrated our parametric hash function into the CACTI tool [Muralimanohar *et al.* (2009)] and have run experiments for a set of configurations. Cache sizes considered are 4KB, 8KB, 16KB, 32KB and 64KB; associativities are 1-way (direct-mapped), 2-way, 4-way, 8-way and fully-associative. Cache line size has no meaningful impact, so we have considered 16 bytes per line as in the rest of the evaluation. Other relevant parameters are technology node (32nm), type of transistors (low operating power ones), access mode (sequential tag and data access for low power) and ports (1 read/write port).

Table 5.5: Relative energy increase of (i) random placement (top rows) and (ii) fully-associative (bottom rows) caches w.r.t. modulo placement cache designs

	8-way RP vs base	**4-way RP vs base**	**2-way RP vs base**	**DM RP vs base**
4KB	6.8%	7.7%	8.3%	8.3%
8KB	4.8%	5.4%	5.6%	5.7%
16KB	3.4%	3.5%	3.6%	3.6%
32KB	2.2%	2.3%	2.3%	2.4%
64KB	1.5%	1.5%	1.5%	1.5%
	FA vs 8-way base	**FA vs 4-way base**	**FA vs 2-way base**	**FA vs DM base**
4KB	588.3%	680.3%	743.2%	743.6%
8KB	822.8%	932.1%	959.5%	981.9%
16KB	1199.0%	1220.7%	1254.8%	1272.2%
32KB	1559.4%	1608.6%	1630.2%	1662.3%
64KB	2039.0%	2074.4%	2107.0%	2085.3%

Table 5.5 shows the relative energy overhead per read access for all non-FA configurations implementing RP with respect to the design without the parametric hash function, whose energy consumption ranges between 3.8pJ and 22.2pJ per access for different configurations. Also, the overhead of a FA cache implementing random replacement is shown with respect to the same baseline (its energy consumption ranges between 32.9pJ and 474.3pJ per access for different configurations). We observe that the energy overhead per read access ranges between 1.5% and 8.3% across all configurations for our RP cache (around 0.3pJ). Since such parametric function is independent of the cache size, the larger the cache size, the lower the relative energy overhead is. This overhead is largely below that incurred by a MBPTA-friendly FA cache whose energy overhead is between 588% and 2107%.

Analogously, Table 5.6 shows the relative access time increase per read access. The relative access time increase for our RP cache is between 28% and 54% (around 0.2ns), being lower for larger caches since the delay of the parametric hash function is almost independent of the cache size, whereas delay for caches increases from around 0.37ns (4KB) to around 0.71ns (64KB). In fact, for sufficiently large caches fewer XOR levels may be needed, thus further reducing the relative delay of the RP caches. Instead, FA caches have much higher access times for 8KB caches and above (between 1.53ns and 67.27ns), and a bit higher average access time for tiny 4KB caches (0.59ns). Overall, the access time may only impact hit latency, which, as stated before, is not the dominant factor in WCET.

Table 5.6: Relative access time increase of (i) random placement (top rows) and (ii) fully-associative (bottom rows) caches w.r.t. modulo placement cache designs

	8-way RP vs base	**4-way RP vs base**	**2-way RP vs base**	**DM RP vs base**
4KB	54.4%	52.7%	49.1%	52.6%
8KB	49.3%	45.8%	45.6%	45.6%
16KB	39.2%	40.5%	40.5%	40.5%
32KB	34.6%	34.6%	34.6%	34.7%
64KB	28.3%	28.3%	28.4%	29.2%
	FA vs 8-way base	**FA vs 4-way base**	**FA vs 2-way base**	**FA vs DM base**
4KB	60.3%	55.5%	44.8%	55.1%
8KB	275.6%	248.6%	247.1%	247.1%
16KB	840.8%	872.8%	872.8%	872.8%
32KB	2918.9%	2919.2%	2919.3%	2929.7%
64KB	9365.1%	9367.3%	9396.9%	9664.9%

Overall, our RP cache is a much more efficient design than exotic fully-associative caches enabling MBPTA. Furthermore, the larger the cache size, the lower the relative overheads introduced by our parametric hash function. Thus, our design is both efficient and scalable.

5.5 External Results

In addition to the results obtained in the scope of this thesis and presented so far in this Chapter, we have evaluated random placement random replacement designs in the the context of the FP7 projects PROARTIS and PROXIMA in collaboration with industrial partners.

In particular in [Wartel *et al.* (2013)] an avionics case study has been evaluated on top of the simulation environment we have developed in this thesis and described in Chapter 3, implementing random placement and replacement caches and TLBs and the MBPTA-compliant architecture described in Chapter 4. This work demonstrates the scalability of MBPTA in large scale applications deployed in IMA systems and the fitness of time-randomised caches in industrial environments.

The results indicate that although the average performance of RP-RR caches is lower than the one of deterministic caches, the degradation in average performance of the application is less than 1%. However, the most important outcome is that compared with the current industrial practice of 20% inflation factor over the maximum observed execution time (MOET), enabling the use of MBPTA with time-randomised caches allows the computation of tighter pWCET values, with a scientifically sound method.

In the context of PROXIMA project, our hardware proposals have been implemented in LEON 3 RTL – a CRTES processor extensively used in the aerospace domain – and are synthesised on an a Terasic DE4 FPGA [Hernandez *et al.* (2015)]. This work demonstrates the feasibility of our proposals to be implemented in real hardware designs, to build MBPTA-compatible processors. The evaluation of this design with EEMBC benchmarks also indicate a difference within 1% of the deterministic designs. Furthermore, a space case study from the European Space Agency has been evaluated on this platform showing a negligible performance difference compared to the deterministic platform [Fernandez *et al.* (2017)].

5.6 Related Work

WCET impact of caches has been studied extensively [Reineke *et al.* (2007)], including several levels of cache [Lesage *et al.* (2009)] and locking mechanisms [Puaut & Decotigny (2002)] to increase predictability and hence, provide tighter WCET.

Some embedded processors already implement random replacement policies on set-associative caches [ARM (2006)] [Cobham Gaisler (2011)]. Randomised caches in high-performance processors have been proposed to remove cache conflicts by using pseudo-random hash functions [Topham & González (1999)] [Seznec & Bodin (1993)]. However, the behaviour of all those cache designs is fully deterministic, and therefore, whenever a given input set produces a pathological access pattern, it will happen systematically for such input set. Therefore, although the frequency of pathological cases is reduced, they can still appear systematically because there is no way to prove that their probability is bound.

Some work on PTA has been done based on the assumption that execution times are truly i.i.d. and that frequencies for execution paths provided by the user match actual probabilities of those paths [David & Puaut (2004)]. Later work has shown how to perform PTA with no assumption on the probabilities of execution paths and how to use random caches in PTA systems [Cazorla *et al.* (2013a)] [Cucu-Grosjean *et al.* (2012)]. Concretely, authors showed that randomised replacement effectively avoids pathological behaviour of deterministic replacement policies while achieving reasonable performance. Some authors have tried to perform PTA on top of conventional cache designs [Liang & Mitra (2008)]. Unfortunately, this can only be done if the user is able to provide the *true probability* (not the frequency) of each cache layout and each execution path to occur for *all instances* of the system deployed, which is, in general, unattainable.

To the best of our knowledge, our work is the first enabling the use of the most common and efficient cache designs, i.e. set-associative and direct-mapped caches, in probabilistically analysable CRTES while preserving the properties needed by sound PTA techniques [Cazorla *et al.* (2013a)] [Cucu-Grosjean *et al.* (2012)].

5.7 Summary

MBPTA enables affordable analysis of complex hardware in safety-critical real-time systems by reducing the amount of information about the hardware and software state required to provide trustworthy WCET estimates. Yet, MBPTA relies on some properties that existing hardware fails to provide. In particular MBPTA requires that the execution times of the program on the target platform can be modelled with i.i.d. random variables and that execution conditions during analysis match or upperbound those at operation.

In the case of the cache, the deterministic behaviour of placement and replacement policies makes it impossible to attach a true probability to different execution times or guarantee same execution conditions at both analysis and operation. Only unaffordable fully-associative caches with random replacement would allow deriving true probabilities, which hold the same at both analysis and operation independently of the memory layout. This chapter presents the first random placement policy based on a parametric hash function so that i.i.d. execution times are obtained, thus enabling the use of efficient set-associative and direct-mapped caches in the context of probabilistic timing analysis. We further show that our cache design can be implemented with little overhead in terms of complexity, energy and performance.

While in this chapter we have focused on devising random placement and replacement policies and implementations for first level caches, in the next Chapter we extend random placement policies to arbitrary complex memory hierarchies, with multiple levels of caches.

Chapter 6

Multiple Level Hardware Time-Randomised Caches

6.1 Introduction

In this chapter we extend our work on single-level random placement-random replacement caches introduced in the previous chapter, and we analyse for the first time in the literature the worst-case timing behaviour of multi-level time-randomised caches. Our analysis, which scales to an arbitrary number of cache levels, covers unified data and instruction caches, different write, write-allocation and inclusion policies among the different levels. Our analysis builds upon some of the properties of time randomised caches. First, the particular memory address of an access does not determine the cache set in which it is mapped since the placement in cache is time randomised (Chapter 5) and second, the hit/miss probability of a given access to an address $@_x$ only depends on probabilistic events such as whether the accesses between the current and the last access to $@_x$ miss in cache. Based on these properties, our analysis identifies those events that cause accesses to the different cache levels as well as their probability of occurrence.

We evaluate 2-level time-randomised cache setups with a unified second level cache (shared among data and instructions). We consider inclusive and non-inclusive caches, as well as write-through and write-back first level caches. Our results prove that (1) multi-level time-randomised caches fulfil the requirements of measurement-based PTA (MBPTA) [Cucu-Grosjean *et al.* (2012)]. We also show how multi-level time-randomised caches decrease execution time by 30% on average. Reduction in terms of WCET estimates is even higher (55% on average) since multi-level caches also reduce the probability of pathological cache behaviour resulting in large execution times, hence execution time variability, with respect to single-level cache setups.

6.2 Cache Characteristics and Assumptions

In a multi-level cache design, *inclusivity* of the lower cache levels (those closer to the cores) 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[1]. This implies that, whenever a cache line is evicted from the upper level cache, all cache lines in the lower level cache holding some or all contents of the cache line evicted in the upper level cache, are also evicted. When *exclusivity* is applied[2], 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. *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.

Upper cache levels can be either shared among data or instructions or kept private. While private caches have been regarded as easier to analyse, unified (shared) ones are the most common choice due to their lower overheads. Thus, unlike previous works, we enable for the first time the analysis of unified upper cache levels storing data and instructions.

Write operations introduce complexities in the behaviour of the cache that are handled with a cache-write policy and a write allocation policy. There are two main write policies, namely, *write-through* (WT) and *write back* (WB). In the former, write operations occur in the current cache and are forwarded to the next cache level so that both caches hold consistent data. In WB caches, write operations occur only in the lower level cache, and the update of the next level is postponed until the cache lines containing the *dirty* data are evicted from the lower level cache. There are two write allocation policies. With *write allocate* (WA), on a write miss, data are fetched into cache, as it is the case for read misses, and, once fetched, the write operation occurs. With *no-write allocate* (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, since they are the most common choices. Though, nothing prevents our analysis to be extended to other combinations.

[1]Note that contents may not be up-to-date in both caches if write operations are not propagated immediately as explained later, but at least an older version of the data is in place in both caches.

[2]We use the term exclusive cache rather than non-inclusive, because the latter just implies that inclusivity is not controlled, which is different than requiring exclusivity.

Table 6.1: Events in a NIC cache hierarchy with WT-nWA L1 and WB-WA L2

Event id	latency	L1	L2	L2 dirty?	actions	probabilities
1)	Lat_{ld1}	L1 ld hit			(a) Send data from DL1 to the core	$P^{@A}_{L1,hit}$
2)	Lat_{ld2}	L1 ld miss	L2 ld hit		(b) Send data from UL2 to DL1 and to the core	$P^{@A}_{L2,hit} \times P^{@A}_{L1,miss}$
3.1)	Lat_{ld3}		L2 ld miss	L2 dirty	(c) Write dirty line to mem, (d) load new line into L2 and (b)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,dirty}$
3.2)	Lat_{ld4}			L2 clean	(d) and (b)	$P^{@A}_{L2,miss} \times P^{vctm}_{L1,miss} \times P^{vctm}_{L2,clean}$
1)	Lat_{st1}	L1 st hit	L2 st hit		(e) write data into L1 and (f) write data into L2	$P^{@A}_{L2,hit} \times P^{@A}_{L1,hit}$
2.1)	Lat_{st2}		L2 st miss	L2 dirty	(c), (d), (e) and (f)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,hit} \times P^{vctm}_{L2,dirty}$
2.2)	Lat_{st3}			L2 clean	(d), (e) and (f)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,hit} \times P^{vctm}_{L2,clean}$
3)	Lat_{st4}	L1 st miss	L2 st hit		(f)	$P^{@A}_{L2,hit} \times P^{@A}_{L1,miss}$
4.1)	Lat_{st5}		L2 st miss	L2 dirty	(c), (d) and (f)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,dirty}$
4.2)	Lat_{st6}			L2 clean	(d) and (f)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,clean}$

6.3 Time Randomised Multi-level Caches

In this section, we focus first on random-replacement fully-associative multi-level caches for the sake of clarity. In Section 6.3.3 we extend our analysis to random placement featured in set-associative and direct-mapped caches. With the same aim and without loss of generality we focus on a 2-level cache hierarchy. In the first level we find an instruction (IL1) and a data cache (DL1). In the second level we have a unified L2 cache (UL2). We generalise our analysis for more than 2 levels also in section 6.3.3. We consider a WT-nWA DL1 and a WB-WA UL2 caches, as this is a very common organisation and allows us reasoning about both types of cache-write and write allocation policies. Note that cache-write and write allocation policies are irrelevant for IL1, as its contents are read-only. We provide the analysis for both non-inclusive and inclusive caches, as they are the common case. Later, in Section 6.5, we also describe the case of exclusive caches. Considerations related to the hardware implementation are described in section 6.3.4.

In the remaining of this section we refer to the probability of an $<event>$, such as a hit or a miss, in each cache level as: $P^{<op>}_{<L>,<event>}$, where L is the cache level, i.e. IL1, DL1 or UL2, $<op>$ is the type of access, i.e. load (ld) [1] and store (st), or the address of the access when the type of operation does not affect its probability. Each probability is provided under a cache configuration that determines the write, allocation and inclusivity policies.

For computing the probabilities in the probability vector of the ETP of every memory operation, we consider the following sequence of accesses: $< I_0^{mop@A} \quad I_1^{ld1} \quad I_2^{st1} \quad I_3^{ld2} ... \quad I_r ... \quad I_s^{stm} ... \quad I_t^{ldn} \quad I_{t+1}^{mop@A} >$, where the subindex is the instruction id, the superindex indicates the number of load and store operation, and where $I_{t+1}^{mop@A}$ is the memory operation whose hit/miss probability we

[1] A load can be an instruction load sent by the instruction cache or a data load sent by the data cache.

Table 6.2: Events in an inclusive cache hierarchy with WT-NWA L1 and WB-WA L2 and their associated probabilities

Event id	latency	L1	L2	L2 dirty?	actions	probabilities
1)	Lat_{ld1}	L1 ld hit			(a) Send data from L1 to the core	$P^{@A}_{L1,hit}$
2)	Lat_{ld2}	L1 ld miss	L2 ld hit		(b) Send data from UL2 to L1 and (a)	$P^{@A}_{L2,hit} \times P^{@A}_{L1,miss}$
3)			L2 ld miss		(c) Check inclusivity of evicted line	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss}$
3.1)	Lat_{ld3}			L2 dirty	(d) write dirty line to mem, (e) load new line into L2, (b) and (a)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,dirty}$
3.2)	Lat_{ld4}			L2 clean	(e), (b) and (a)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,clean}$
1)	Lat_{st1}	L1 st hit			(f) write data into L1, (g) write data into L2	$P^{@A}_{L1,hit}$
2)	Lat_{st2}	L1 st miss	L2 st hit		(g)	$P^{@A}_{L2,hit} \times P^{@A}_{L1,miss}$
3)			L2 st miss		(c)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss}$
3.1)	Lat_{st3}			L2 dirty	(d), (e) and (g)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,dirty}$
3.2)	Lat_{st4}			L2 clean	(e) and (g)	$P^{@A}_{L2,miss} \times P^{@A}_{L1,miss} \times P^{vctm}_{L2,clean}$

want to derive for each cache organisation. Note that *mop* stands for any memory operation (either *ld* or *st*). We use it in those cases where the particular type of memory operation being considered is irrelevant. The load and store operations between the accesses to $@_A$ access different cache lines in DL1, IL1 and UL2 to those where $@_A$ and the instruction loading $@_A$ are stored.

6.3.1 No Inclusivity Control (NIC)

When no inclusivity control is used among the different cache levels, the hit accesses and the evictions carried out in one level have no impact on previous or next cache levels. Table 6.1 shows the different events in each level of the cache, the actions taken in the cache on that event and the associated probability.

When a load access hits in DL1 (1), which happens with a probability $P^{@A}_{DL1,hit}$, data are sent to the core. In case of miss in DL1 and hit in UL2 (2), data are sent from UL2 to DL1 and the core. Given that the RIIs used for each cache are different as explained later, their placement functions are different and thus, the events 'hit in DL1' and 'hit in UL2' are independent. Hence, the probability of both events to occur can be obtained by multiplying their respective probabilities. In case of a miss in both DL1 and UL2 (3), data are loaded from memory to UL2 and DL1, and sent to the core. If the line evicted is dirty[1] (3.1), it is written back to memory before the new line is loaded from memory to UL2. If it is clean (3.2) no line is written back. Note that the instruction cache (IL1) events are the same as the load events for the DL1.

Stores update DL1 when they hit and are always forwarded to UL2. If they hit in UL2, UL2 is updated (1). In case of miss in UL2 (2), first a cache line

[1]Write operations in a WB cache make cache lines to be inconsistent with upper levels in the memory hierarchy, so on an eviction their contents must be updated in upper levels. Those lines are referred to as *dirty lines*.

(victim) is selected for eviction and, if it is dirty (2.1), which happens with a probability $P_{L2,dirty}(vctm)$, it is written back to memory. Then, the new line is fetched from memory into UL2 and it is updated with the data carried out by the store operation. If the line was clean (2.2), the same actions, but writing the victim to memory, are carried out.

In case of miss in both DL1 and UL2 (3), the line is written into UL2, but not brought to DL1. In case of miss in both (4.1 and 4.2), the evicted line is written to memory in case it is dirty, the new line is brought into UL2 and updated with the new data.

The most important appreciation from Table 6.1 is that *all the events that may potentially happen in all caches have an associated probability* ($P_{UL2,hit/miss}(@)$, $P_{IL1,hit/miss}(@)$, $P_{DL1,hit/miss}(@)$ and $P_{UL2,dirty}(@)$). Next, we derive how those probabilities can be approximated. Note, however, that, assuming that core operations can be analysed probabilistically, the fact that cache events are probabilistic makes the application of MBPTA correct, since EVT makes no assumption on the particular probability distribution function of the events under consideration or on whether they depend on each other.

A.1. $P_{DL1,miss}(@)$. The miss probability of an access to $@_A$ in a WT-nWA DL1, $P_{DL1,miss}(@_A)$, with a WB-WA UL2 with no inclusivity control is affected by the intermediate accesses carried out in between that access to $@_A$ and the previous access to $@_A$. In particular, $P_{DL1,miss}(@_A)$ is given by the number of memory operations between both accesses to $@_A$, and the probability of miss of each access. The higher the number of accesses and their respective miss probabilities, the higher the probability of miss of the second access to $@_A$, which can be approximated as:

$$P_{DL1,miss}(@_A) = 1 - \left(\frac{W_{DL1} - 1}{W_{DL1}}\right)^{\sum_{i=1}^{n+m} P_{DL1,miss}(DL1acc_i)}$$

where W_{DL1} is the number of ways in DL1[1], $n+m$ is the number of intermediate loads and stores, and $P_{DL1,miss}(DL1acc_i)$ their associated miss probabilities.

A.2. $P_{UL2,miss}(@)$. The probability of missing in UL2 for a given access to address $@_A$ (regardless of whether it is a data or instruction address) is given by the number of accesses performed to UL2 between the current access and the previous access to $@_A$. This includes both, UL2 instruction accesses coming from the IL1 and UL2 data accesses coming from DL1 as shown in the formula below, where W_{UL2} is the number of lines in the UL2.

[1]Recall that in this section we focus on fully-associative caches, so the number of cache ways of the cache is also the number of cache lines.

$$P_{UL2,miss}(@_A) = 1 - \left(\left(\frac{W_{UL2}-1}{W_{UL2}} \right)^{\sum_{i=1}^{t} P_{miss,IL1}(I_i) \times P_{miss,UL2}(I_i)} \times \left(\frac{W_{UL2}-1}{W_{UL2}} \right)^{\sum_{i=1}^{n+m} P_{miss,DL1}(mop_i) \times P_{miss,UL2}(mop_i)} \right)$$

The exponent in the first element in the formula above accounts for the effect of instruction misses (between both accesses to $@_A$) in the IL1 (and hence accesses to the UL2) that also miss in UL2. The exponent in the second element is the probability of a memory operation to miss in the DL1 and UL2. We can simplify the above formula as $P_{UL2,miss}(@_A) = 1 - \left(\frac{W_{UL2}-1}{W_{UL2}} \right)^{\sum_{i=1}^{k} P_{UL2,miss}(L2acc_i)}$, where the exponent is the accumulated miss probability of all UL2 accesses between the current and the previous access to $@_A$.

A.3. $P_{UL2,dirty}$. When $@_A$ misses in UL2, a random line in the corresponding set is selected for eviction. There is a probability that the selected line is dirty. While hit and miss probabilities only depend on the accesses in between the current and the previous accesses to a particular address, $P_{UL2,dirty}$ depends on all past accesses since the beginning of the execution of the program (assuming an initial empty cache state). Therefore, and only for approximating $P_{UL2,dirty}$ we consider the following sequence: $<I_0^{mop@_0}, I_1^{mop@_1}, ..., I_{i-1}^{mop@_{i-1}}, I_i^{mop@_A}>$, in which we assume that $I_i^{mop@_A}$ misses in cache, resulting in a cache line being randomly evicted. We obtain $P_{UL2,dirty}(I_i)$, that is, the probability of a dirty line to be evicted when I_i is executed, as the fraction of dirty lines in the cache set where $@_A$ is when the second access to $@_A$ occurs. In other words, the accumulated probability that each line in that set has not been evicted from cache since it was last accessed by a store operation.

$$P_{UL2,dirty}(I_i) = \sum_{j=0}^{i-1} P_{surv}^{dirty}(@_j, i)$$

In the equation above, $P_{surv}^{dirty}(@_j, i)$ is the probability of instruction I_j to leave a dirty line in cache and this line to be still present when instruction I_i (the one accessing $@_A$) is executed. It is defined as follows:

$$P_{surv}^{dirty}(@_j, i) = \begin{cases} 0 & \text{if } isload(I_j) \\ \left(\frac{W_{L2}-1}{W_{L2}} \right)^{\sum_{k=j+1}^{i-1} P_{miss}(@_k)} & \text{if } isstore(I_j) \end{cases}$$

In the equation above, $isload(I_k)$ is true when I_k is a load instruction, similarly $isstore(I_k)$ is true when I_k is a store. Lines dirtied by stores survive with a given probability that depends on the number of misses between them and the access to $@_A$.

6.3.2 Inclusive Caches

The main difference between inclusive caches with respect to caches without inclusivity control (NIC) is that UL2 evictions may require evicting some cache lines in DL1. If a line being evicted from UL2 is present in DL1, it is also evicted from DL1. Note that inclusivity is typically deployed only for data caches since this simplifies hardware design to deal with either write operations in write-through caches or dirty lines evicted in write-back ones. Instruction caches typically do not support any type of write operation (other than filling cache lines when fetched), so there is no need for making them inclusive.

Table 6.2 shows the different events that may happen to a cache access and their associated probability. On an UL2 miss, a victim is selected to be evicted. In addition to checking whether it was dirty, in which case it is written back to memory, it must be checked whether that line is present in DL1 (in fact in all L1 inclusive caches), in which case it is invalidated from the corresponding L1 cache. As DL1 is WT, the invalidation consists simply in setting an 'invalid' bit (no further transaction is initiated). Therefore, we consider that the latency of checking for invalidations is the same regardless of whether a line is finally invalidated. How to deal with WB caches and dirty lines is later described in Section 6.3.3. We also observe that in inclusive caches the event 'DL1 hit and UL2 miss' is not possible, since all DL1 contents are also present in the UL2.

As for NIC caches, the events that may potentially happen in the different caches have an associated probability ($P_{UL2,hit/miss}(@)$, $P_{IL1,hit/miss}(@)$, $P_{DL1,hit/miss}(@)$ and $P_{UL2,dirty}(@)$). The value of some of those probabilities change with respect to the NIC case as detailed next.

B.1. $P_{DL1,miss}(@)$. The probability of miss in DL1 of an access to $@_A$, $P_{DL1,miss}(@_A)$, with an inclusive UL2 is affected by the accesses carried out in between that access to $@_A$ and its previous access, and the probability of miss of those intermediate accesses. There are two types of accesses to the DL1 that can happen. First, *memory accesses (MA)* between the two accesses to $@_A$ sent from the core to the DL1. And second, *Data Inclusivity Requests (DIR)* sent from the UL2 to DL1 due to the UL2 accesses between the two accesses to $@_A$ that evict lines from UL2, thus causing a subsequent eviction in DL1 of the line evicted from UL2.

The number of data memory accesses sent to UL2 from the core are given by $P_{MAmiss} = \sum_{i=1}^{k} P_{DL1,miss}(mop_i)$, where mop_i are the loads and stores in between

the two accesses to $@_A$ and $P_{DL1,miss}(mop_i)$ is the miss probability of each access computed as for single-level caches. Analogously, instruction accesses sent to UL2 are given by $P_{Imiss} = \sum_{i=1}^{k} P_{IL1,miss}(I_i)$, where I_i stands for any instruction in between the two accesses to $@_A$ and $P_{IL1,miss}(I_i)$ is the miss probability of each such instruction computed as for single-level caches.

When a data or instruction access between two accesses to $@_A$ causes a UL2 miss, this generates a UL2 eviction which can evict $@_A$ from UL2, which would imply removing $@_A$ from DL1 to keep inclusivity. The number of DIRs is given by number load and store operations between two accesses to $@_A$ that miss in DL1 plus the number of instructions fetched between those two accesses to $@_A$ that miss in IL1. Note that though the UL2 is inclusive of DL1, hits to DL1 are ensured to also hit in UL2, hence not generating any eviction that could evict the line in UL2 where $@_A$ is. This is particularly important for store operations that access UL2 regardless of whether they hit DL1. We approximate the accumulated probability due to inclusivity evictions as follows:

$$P_{DIRev} = \sum_{i=1}^{n+m}\left(P_{DL1,miss}(mop_i) \times P_{UL2,miss}(mop_i) \times \frac{1}{W_{UL2}}\right) + \sum_{i=1}^{t}\left(P_{IL1,miss}(I_i) \times P_{UL2,miss}(I_i) \times \frac{1}{W_{UL2}}\right)$$

where $n+m$ is the number of loads and stores and t the number of instructions between both accesses to $@_A$. The first element in the first row of the equation is the probability that any data access in between two accesses to $@_A$ misses in the data cache. The second element is the probability that those DL1-missing accesses, that access the UL2, miss in the UL2. The last element is the probability that each evicted L2 cache line contains $@_A$. The second row of the formula is analogous for IL1.

Overall, the miss probability of $@_A$ in DL1 can be approximated as $P_{DL1,miss}(@_A) = 1 - \left(\frac{W_{DL1}-1}{W_{DL1}}\right)^{P_{MAmiss}+P_{DIRev}}$.

B.2. $P_{UL2,miss}(@)$. There is a probability that an access to $@_A$ in the UL2 (regardless of whether it is a data or an instruction access), and so its $P_{UL2,miss}(@_A)$, is affected by the inclusivity policy. An instruction with instruction address (i.e. Program Counter) $@_A$ accesses UL2 if it misses in IL1. Since DL1 contents are included in UL2, the access $@_A$ cannot hit in those W_{DL1} lines of the UL2 keeping the contents of the DL1. Similarly, a DL1 miss cannot hit in the UL2 lines keeping the contents of DL1. Hence, $@_A$ can only hit in $W_{UL2} - W_{DL1}$ lines, which we call $W_{UL2nonDL1}$ being its miss probability approximated by:

$$P_{UL2,miss}(@_A) = 1 - \left(\frac{W_{UL2nonDL1} - 1}{W_{UL2nonDL1}}\right)^{\sum_{i=1}^{k} P_{UL2,miss}(L2acc_i)}$$

where the exponent is the miss probability of the accesses between $@_A$ and its last access.

B.3. $P_{UL2,dirty}$. It is not affected by the inclusivity control, so it remains as described for the NIC case.

6.3.3 Generalising the Latency/Probability Cache Model

For the sake of simplicity, we have assumed that all accesses in the different sequences we have used to describe the hit/miss probability of each event, i.e. $< I_0^{mop@A} \quad I_1^{ld1} \quad I_2^{st1} \quad I_3^{ld2} ... \quad I_r ... \quad I_s^{stm} ... \quad I_t^{ldn} \quad I_{t+1}^{mop@A} >$ and $< I_0^{mop@_0}, I_1^{mop@_1}, ..., I_{i-1}^{mop@_{i-1}}, I_i^{mop@_A} >$, go to a different cache line each, but the first and last access to $@_A$ that access the same line. Considering the case in which intermediate accesses may access the same cache line addresses just makes the computation of probability approximations more complex, since events are probabilistically dependent; however, *the events affecting the timing behaviour of the cache are still probabilistic as required for the application of MBPTA* (see Section 6.4.) Recall, that MBPTA makes no assumption on the probability distribution function of any random event. Note also, that the probabilities computed assuming that accesses are assumed to go to different addresses represent an upper-bound of the actual probabilities when they may go to the same address line. This is so, because when two accesses go to the same address, their reuse distance reduces and so do their miss probabilities.

Besides that, there are several dimensions in which our model can be generalised. First, random placement since in our analysis above we consider only random replacement. Second, different cache line sizes between different levels. Third, considering more than 2 cache levels. And fourth, different inclusivity arrangements between different cache levels.

Random Placement. Random placement requires taking into account the probability that any of the k different (unique) accesses between the two accesses to $@_A$ access the set where $@_A$ is placed. This is given by the formula $\left(\frac{S-1}{S}\right)^k$, where S is the number of sets. As we have done for the random replacement in previous sections, for each cache level we can compute the number of those potential accesses. For the IL1 these are the number of unique instruction requests between the two accesses to $@_A$. For the DL1 we have the number of unique memory operations plus the number of unique data inclusivity requests. The UL2 is accessed as many times as the number of DL1 and IL1 misses are experienced. The effect of the

placement on the probability of hit of accesses can be multiplied by the effect of replacement computed in previous sections as both are independent, as show in the previous Chapter.

Different cache line sizes. So far we have considered the case where all caches use the same cache line size. However, it may be the case that cache lines in the upper level are larger than those in the lower levels. Let us assume that UL2 lines are q times larger than those of the DL1 and IL1. There are two main ways in which different cache line sizes in each level affect the probability of hit/miss of each access. First, the distribution of accesses on the different cache lines changes. For instance, while accesses $@_B$ and $@_C$ in the sequence $< @_A @_B @_C @_A >$ access different cache lines under a line size setup, they can access the same line if the cache line size is increased. In the latter case, the probability of evicting $@_A$ is smaller since $@_C$ will always hit and hence, will never produce an eviction. This simply reduces the miss probability of $@_A$. Note, however, that if the cache size remains constant, increasing the line size implies reducing the number of sets or ways, which will increase the probability of miss of $@_A$. In any case, hit/miss events remain probabilistic regardless of the cache line sizes and hence, analysable with MBPTA. Second, for some inclusivity control policies, when a dirty line is evicted from UL2, the contents of that line have to be evicted from DL1. If UL2 lines are larger than DL1 ones and hence, contain several DL1 lines, this would produce a potentially larger number of invalidations, reducing the probability of hit of DL1 accesses.

Several cache levels. For setups with several (more than 2) cache levels, the only difference in our analysis consists in taking into account, when analysing a given cache level L_i, the accesses that any other cache level can introduce on L_i. This depends on the inclusivity arrangement selected and the write-miss and allocation policies of each level. This would make the number of events to consider higher, but each event would be still fully probabilistic. As a result, an ETP for each cache access still exists, and hence, MBPTA can be applied.

Inclusivity Arrangements. Similarly, to the previous case, inclusivity policies affect the number of accesses that a given program does to the different cache levels. It also affects, the actual size available in a given level. For instance, when the DL1 is inclusive of the UL2, every miss in DL1 that becomes an access to the UL2, can only hit in the UL2 lines not devoted to keep DL1 information. As we have seen, this can easily be taken into account in the analysis. Analogously, if DL1 is WB and inclusive, dirty lines may be evicted. This may have an impact in latency. To consider this, one should split the case of DL1 evictions into 2 subcases considering whether a dirty line from DL1 is evicted or not. Deriving such probability of dirtiness in DL1 is analogous to the case of UL2.

Overall, the effect of all these variations is purely probabilistic and therefore,

analysable with MBPTA. MBPTA does not need to compute ETPs and hence, it is enough those events to be probabilistic to ensure ETPs exist, which is in turn enough to apply MBPTA.

6.3.4 Hardware Considerations

In random placement multi-level caches it is important guaranteeing that placement choices in every cache are independent to prevent any correlation between the random events in the different caches. This is achieved by simply using different RII values for each cache. Therefore, those addresses conflicting in a particular cache set in a first level cache, thus producing some misses, are very unlikely to be placed in the same set in the second level cache so that the same conflicts do not repeat.

Regarding the overhead of the random placement and replacement caches, it has been shown to be low with respect to modulo-placement LRU-replacement: as shown in the previous chapter the overhead in area and access time is very small and its relative impact further decreases for large caches such as UL2 ones.

6.4 Actual Probabilities

As indicated in Chapter 5, hit/miss probabilities provided in this thesis are an approximation to the actual ones, since MBPTA only requires their existence but not their exact value [Cucu-Grosjean *et al.* (2012)]. To illustrate how actual probabilities can be derived, we use an example where a sequence of accesses $<A_1, B_1, A_2, B_2>$ access cache lines A and B in a fully-associative cache with 4 cache lines. Figure 6.1 depicts the sequence of accesses, the different events that can occur and their probabilities. At the bottom of the figure probabilities for each sequence of events are provided.

A_1 always misses in cache and A is in cache after this access. Then, B_1 misses in cache, but two different cache states may be reached: with a probability of 0.75 B replaces any of the empty lines and with a probability of 0.25 (1 out of 4) B replaces A so that B is the only valid line in cache. A_2 accesses cache and hits if the cache contents were $\{A, B, -, -\}$, which occurs in one of the two different sequences of events at this stage. If A_2 hits (leftmost path in the graph), cache state remains the same ($\{A, B, -, -\}$), otherwise it may happen again that A replaces an empty line (0.75 probability) or B (0.25 probability). Finally, B_2 hits in cache in two of the three different sequences of outcomes.

Overall, if we compute the actual probabilities from the graph, we obtain the probability vectors in Table 6.3. As shown, the probability vector for B_2 differs across the exact computation and the approximation provided by Equation 5.7.

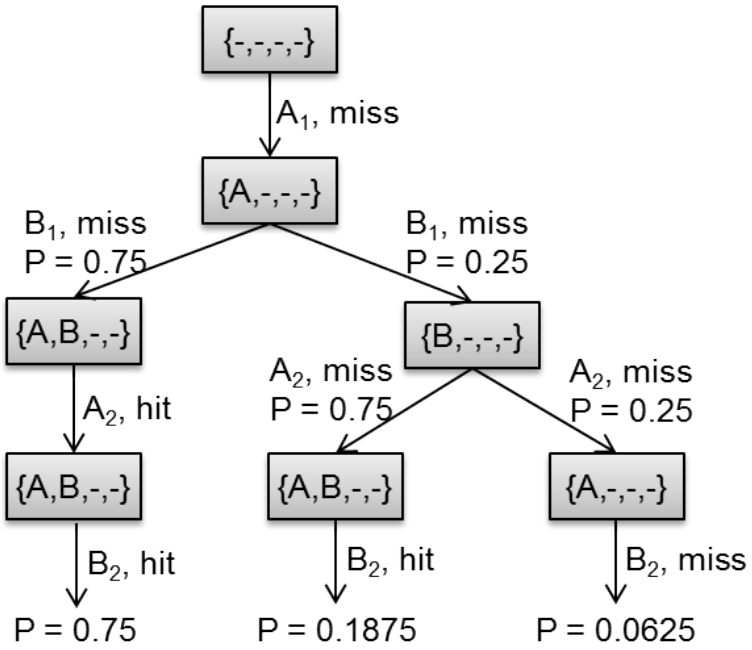


Figure 6.1: Access tree and cache state for the access sequence $<A_1, B_1, A_2, B_2>$.

Table 6.3: Probability vectors for the accesses in the sequence $<A_1, B_1, A_2, B_2>$ for a fully-associative 4-entry

Access	Prob. vector real $\{p_{hit}, p_{miss}\}$	Prob. vector Equation 5.7 $\{p_{hit}, p_{miss}\}$
A_1	$\{0.0, 1.0\}$	$\{0.0, 1.0\}$
B_1	$\{0.0, 1.0\}$	$\{0.0, 1.0\}$
A_2	$\{0.75, 0.25\}$	$\{0.75, 0.25\}$
B_2	$\{0.9375, 0.0625\}$	$\{0.9306, 0.0694\}$

This occurs because the probabilities of hit and miss across different sequences of events differ. However, although those probabilities are not independent, whether a hit or a miss occurs depends solely on random events, which follow exactly the same probability distribution (ETP) at both analysis and operation, so all properties needed by MBPTA [Kosmidis *et al.* (2014d)] are fulfilled.

Note that, as opposed to SPTA [Cazorla *et al.* (2013a)], MBPTA does not need to determine the actual probabilities. SPTA, indeed, needs actual probabilities to be upper-bounded and those probabilities must be independent so that they can be convolved to obtain the probability distribution for the whole program. Obtaining actual probabilities can be done in two main ways: (i) Performing an

Table 6.4: Probabilities of experiencing 0, 1 and 2 hits in the sequence $<A_1, B_1, A_2, B_2>$.

	Real	Convolution
P(0 hits)	0.0625	0.0156
P(1 hit)	0.1875	0.2813
P(2 hits)	0.75	0.7031

'infinite' number of runs and measuring actual probabilities, or (ii) Computing the probability of each particular cache state left by the sequence of hits and misses for previous accesses, and accumulating the probabilities for those cache states where the current cache access would result in a hit/miss, as we do in our example. Unfortunately, even if exact probabilities are obtained, they cannot be convolved, so if SPTA is to be used, a way is needed to compute probability vectors that upper-bound those under any sequence of events for each access, i.e. by making sure that the miss probability used is equal or higher than the miss probability under any sequence of events [Cazorla *et al.* (2013a)].

For the sake of illustration, we provide in Table 6.4 the probabilities of experiencing 0, 1 and 2 hits when executing the sequence $<A_1, B_1, A_2, B_2>$ (i) directly from the probability graph in Figure 6.1 and (ii) by convolving the probability vectors in Table 6.3. For instance, the actual (real) probability of having exactly one hit is 0.1875, which occurs when A_2 misses and B_2 hits. When applying convolutions, such probability of having exactly one hit is obtained as the addition of the probabilities of (1) A_2 missing and B_2 hitting ($0.25 \cdot 0.9375 = 0.2344$) and (2) A_2 hitting and B_2 missing ($0.75 \cdot 0.0625 = 0.0469$), which indeed cannot occur. As expected, probabilities obtained with convolutions neither match nor upper-bound real ones (e.g., SPTA would underestimate the probability of having 0 hits, which is the one leading to the highest execution time).

Overall, deriving actual probabilities would require a more complex formulation than the one we have used in this chapter to derive approximations. The purpose of deriving the probability approximations is proving that hit and miss events occur with a given probability, which is a sufficient condition for enabling the application of MBPTA in multilevel caches.

6.5 Exclusive Caches

In this section we briefly discuss some considerations for exclusive caches. In exclusive caches, contents cannot be replicated across multiple caches. Therefore, on a DL1 and UL2 miss, the new line is fetched from memory to DL1, the line evicted from DL1 is moved to UL2, and the line evicted from UL2 is invalidated (if

clean) or written back to memory (if dirty). Analogously, on a DL1 miss and UL2 hit, the line from UL2 is moved to DL1. One line from DL1 is evicted and placed in UL2, which can also produce a cascade eviction since placement functions in DL1 and UL2 are independent and so, although both lines (the one fetched and the one evicted) are placed into the same set in DL1, they are very unlikely to be placed in the same set in UL2. In general, exclusive caches are not common because of the number of cache line transfers on a DL1 miss.

Exclusive caches are ill-advised in combination with WT policy, as it is the case of the DL1 considered in this work. This is so because on a DL1 store hit, the write operation is sent to UL2, where it is guaranteed to miss due to the exclusivity constraint. This would enforce evicting such particular cache line from DL1 to put it in UL2 (potentially causing a UL2 eviction) or sending the write operation straight to memory, thus jeopardising performance and power due to the increased number of memory accesses.

The events that may potentially happen in the different caches have an associated probability ($P_{UL2,hit/miss}(@)$, $P_{IL1,hit/miss}(@)$, $P_{DL1,hit/miss}(@)$ and $P_{UL2,dirty}(@)$). The values of some of those probabilities change with respect to the non-inclusive and inclusive cases as detailed next.

C.1. $P_{DL1,miss}(@)$. It is not affected by the inclusivity control, so it remains as described for the non-inclusive case, since UL2 cannot produce any eviction in DL1.

C.2. $P_{UL2,miss}(@)$. There is a probability that an access $@_A$ to the UL2, and so its $P_{UL2,miss}(@_A)$, is affected by the exclusivity policy. On a DL1 miss, data can be found in any UL2 line. Since DL1 and UL2 are exclusive, a miss in UL2 occurs if and only if data are not in the $W_{UL2} + W_{DL1}$ lines of DL1 and UL2 together, being its miss probability approximation:

$$P_{UL2,miss}(@_A) = 1 - \left(\frac{W_{UL2} + W_{DL1} - 1}{W_{UL2} + W_{DL1}}\right)^{\sum_{i=1}^{k} P_{UL2,miss}(L2acc_i)}$$

where the exponent is the miss probability of the accesses between $@_A$ and its last access.

C.3. $P_{UL2,dirty}$. It is not affected by the inclusivity control, so it remains as described for the other cases.

6.6 Evaluation

6.6.1 Experimental Framework

We use the simulation environment introduced in Chapter 3 configured with L2 cache enabled. The modelled core is similar to the LEON4 [Cobham Gaisler (2011)] and incorporates bypasses to remove pipeline stalls due to dependences across instructions. During the execution stage, the time-randomised data cache is accessed. We model 4KB, 32-byte line, 4-way set-associative instruction (IL1) and data caches (DL1), both deploying random replacement and random placement.The UL2 is a unified cache keeping data and instructions. It is 128KB with 32-byte lines and 8-way set associativity. The UL2 access latency is 10 cycles and the latency to access memory 100. The DL1 deploys WT-nWA policies and the UL2 is WB-WA.

We use several inclusivity arrangements: a first setup where we make DL1 inclusive of the UL2 (*L1-L2inc*) and a second setup in which we do not exercise any inclusivity policy (*L1-L2nic*). We also evaluate the effect of making the DL1 write-back when inclusivity is exercised (*L1-L2wb*). DL1 and IL1 caches are connected to the UL2 through fully-dedicated bidirectional buses, whose access latency can be bounded using the technique presented in [Paolieri *et al.* (2009a)].

The objective of our analysis is to effectively reduce the pWCET estimates that can be obtained for programs when several levels of cache are used. Of course, only on those cases in which the average execution time of the program reduces when several levels of cache are deployed, we can expect some reduction in the pWCET. As a reference point we use a setup with a single level of cache in which DL1 and IL1 have the same size they have in the other setups, i.e. 4KB.

For the evaluation we use the simulation framework and the EEMBC Autobench benchmark suite, described in Chapter 3.

6.6.2 Compliance with MBPTA requirements

On the one hand, the core architecture presented in previous section has been shown to be MBPTA compliant (Chapter 4). On the other hand, we can derive a probability of every event affecting the timing behaviour of a cache access (see Section 6.4), or approximate such probabilities (see Section 6.3) . As a result, for each instruction an ETP exists, which makes our multi-level cache processor architecture MBPTA-compliant by construction. This is so, because (1) the latency of each instruction can be modelled with i.i.d. random variables, and (2) the execution of a sequence of instructions leads to another ETP, i.e. random variable, which at the coarsest granularity level represents the ETP of the program.

Table 6.5: Independence and identical distribution tests results (outcome independence test / outcome i.d. test).

Benchmarks	L1-L2 INC	L1-L2 NIC	L1-L2 WB	L1 (only)
a2time	0.03/0.29	0.83/0.41	0.46/0.44	0.90/0.49
aifftr	0.71/0.74	0.95/0.33	0.82/0.59	1.19/0.33
aifirf	0.40/0.11	1.04/0.20	0.13/0.94	1.04/0.79
aiifft	0.68/0.32	0.50/0.41	0.96/0.17	1.09/0.94
cacheb	0.63/0.93	1.11/0.72	1.20/0.35	0.79/0.66
canrdr	0.79/0.16	0.75/0.54	1.00/0.37	0.32/0.91
iirflt	0.96/0.85	0.68/0.41	0.78/0.50	0.07/0.22
puwmod	1.39/0.67	0.99/0.25	0.94/0.75	0.30/0.71
rspeed	0.47/0.43	1.33/0.51	0.91/0.24	1.35/0.42
tblook	1.33/0.92	0.52/0.86	0.34/0.26	0.76/0.44
ttsprk	0.19/0.43	0.89/0.52	0.21/0.42	0.67/0.63

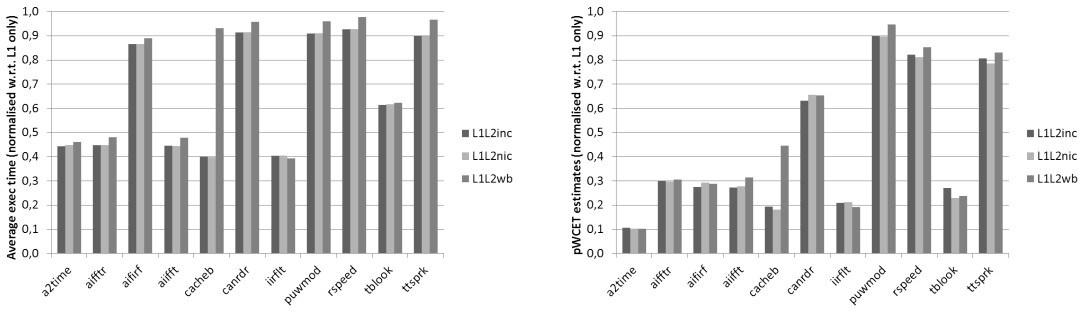
In order to show that the execution times of the program fulfil i.i.d. requirements, we use the tests described in Chapter 3, similar to [Cucu-Grosjean *et al.* (2012)]. Table 9.1 shows the results of both tests for all EEMBC benchmarks under all multi-level cache configurations. As expected, both tests are passed in all cases so independence and identical distribution hypotheses cannot be rejected. This statistical results reinforce the probabilistic analysis we did in Section 6.3 on the timing behaviour of multi-level caches.

6.6.3 Reduction in pWCET Estimates

We consider the same exceedance probability of 10^{-15} per run as in the evaluation of the single level caches in the previous Chapter (Section 5.4.6).

The objective of our analysis is to effectively enable the use of multi-level caches such that significant reductions can be obtained in the pWCET estimates derived by MBPTA. The reduction that can be obtained depends on each application and in the particular use of cache that the application does. Applications requiring little cache space are very unlikely to benefit from having a UL2 cache in place in terms of average performance and pWCET estimates.

Figure 6.2a shows the average performance that each EEMBC obtains when the different cache setups are deployed. All results are normalised to the *single-level* cache setup. We observe that some benchmarks are quite insensitive to having a two-level cache hierarchy achieving a small execution time reduction. Those benchmarks – `aifirf`, `canrdr`, `puwmod`, `rspeed` and `ttsprk` – achieve an average performance reduction in the range 5%-15%. This is mainly due to the fact that those benchmarks have a small code footprint and data working set that fit in



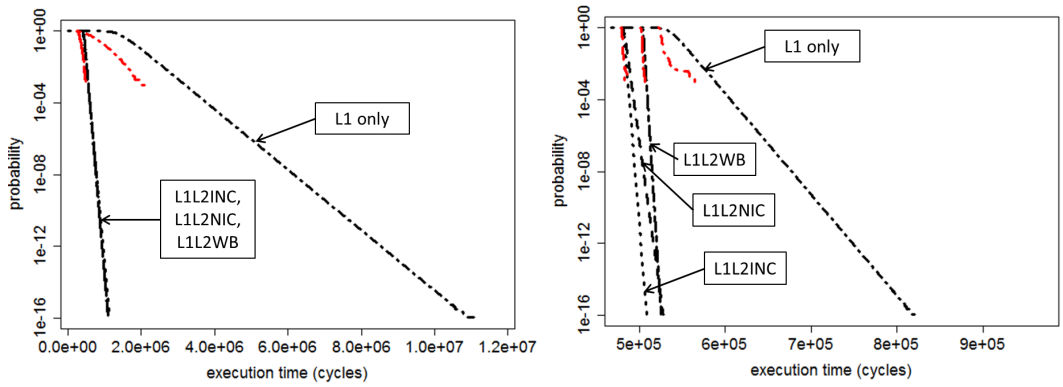
(a) Average execution time (b) pWCET estimates

Figure 6.2: (a) Average and (b) pWCET execution time for different cache configurations normalised to the single-cache level setup

IL1 and DL1 respectively. The rest of the benchmarks significantly benefit from having a UL2 cache. `a2time`, `aifftr`, `aiifft`, `cacheb`, `iirflt` and `tblook` achieve execution time reductions in the range 40%-60%.

It must be also noted that the difference between inclusive and non-inclusive caches is negligible. However, whenever the DL1 is WB and WA, execution time increases. This is particularly noticeable for `cacheb`, whose execution time doubles for the WB-WA configuration. The reason is as follows: when the DL1 is WT-nWA, store instructions can be served without stalling the pipeline. However, under a WB-WA configuration, if store operations miss in DL1, the pipeline is stalled until data are fetched into DL1. In general, this has a relatively low effect on benchmarks whose most store operations hit in DL1, thus not causing any stall. Only few store operations miss in DL1 and increase execution time. However, the store operations in `cacheb` often miss in DL1 and UL2, so they stall the pipeline for long periods of time, thus increasing execution time noticeably.

Figure 10.6 shows the pWCET estimates obtained for every EEMBC benchmark under each cache setup, normalised to the pWCET estimates for the single-level cache setup. pWCET reductions obtained with multi-level caches are more significant than those in terms of average performance. The average performance reduction is around 30% whereas the average pWCET reduction is around 55%. The reason for this behaviour lies on the fact that random placement may map different cache lines to the same set with a relatively high probability in L1 caches, thus causing misses and increasing execution time with non-negligible probability. This effect basically increases the probabilities of high execution times, so MBPTA accounts for that deriving Gumbel distributions [Cucu-Grosjean *et al.* (2012)] with a lower slope, which basically increases pWCET estimates as the exceedance probability decreases. This effect is detailed in the next section through particular examples. On the other hand, whenever a UL2 cache is in place, those



(a) `a2time` (b) `canrdr`

Figure 6.3: pWCET distributions and actual measurements for (a) `a2time` and (b) `canrdr`.

conflicts that may arise in L1 caches in few executions have low impact in execution time because UL2 latency is much lower than that of main memory (10 versus 100 cycles). Moreover, since random placement functions in different caches are independent, L1 conflicts are extremely unlikely to also occur in UL2. Overall, whenever a UL2 cache is used, both execution time and execution time variation decrease, thus leading to much lower pWCET estimates since the Gumbel distribution slope is sharper.

6.6.4 Detailed pWCET Analysis

This section analyses in detail the effect of UL2 caches in pWCET estimates by considering exemplary benchmarks. In particular, we consider `a2time` and `canrdr`. The pWCET estimates we obtain are shown in Figures 6.3a and 6.3b respectively. In those figures we also show complementary cumulative distribution functions (CCDF)[1].

For both benchmarks, the observed execution times (in red and reaching only probabilities down to 10^{-3}) exhibit little variability when the UL2 cache is in place. This leads to sharp slopes for the Gumbel distributions describing the pWCET estimates for those benchmarks. In the case of `a2time` all setups with a UL2 have very similar average performance and pWCET distributions are practically identical. In the case of `canrdr`, execution time when UL2 is used exhibits somewhat higher (still low) variation. This creates some pessimism for pWCET estimates,

[1]The probability distribution function (or PDF) gives the probability of each execution time to occur. The cumulative distribution function (CDF) accumulates probabilities and the complementary CDF or CCDF, is computed as *1-CDF*.

as shown for the L1-L2 NIC setup, whose average performance resembles that of the L1-L2 INC setup, but whose pWCET estimate for an exceedance probability of 10^{-15} per run resembles that of the L1-L2 WB setup.

Finally, we observe how execution time observations for the *single-level* setup for both benchmark are higher and exhibit much higher variability. This leads to a right-shifted pWCET distribution with lower slope and thus, significantly higher pWCET estimates.

6.7 External Results

In addition to the simulation results presented in the previous Section, in the PROXIMA project we had the opportunity to evaluate a certain multi-level cache configuration on a realistic industrial setup. In particular, as a result of industrial cooperation inside PROXIMA, the proposals of this thesis have been implemented in an RTL-implementation of a LEON3 processor [Hernandez *et al.* (2015)] [Kosmidis *et al.* (2016b)].

The implemented memory hierarchy consists of two levels of cache similar to the one presented in Figure 4.3. The L2 cache is unified and it is inclusive of L1 instruction and data caches. The L1 data cache implements a write-through policy, while the L2 is write-back. The size of each L1 cache is 16KB and the size of the L2 is 128KB, each implemented as a 4-way set-associative cache, with random placement and random replacement policy. The processor has 4 cores, but the relevant results for this thesis presented below are obtained in isolation, using the partition feature of the L2 cache, which assigns a cache way to each core. Therefore, each core has exclusive access to a single cache way of 32KB.

For the evaluation, 4 industrial case studies have been used from the CRTES domains of avionics, aerospace and railway [Agirre *et al.* (2016)]. The results vary per application, however in 3 out of 4 case studies the difference between the deterministic setup and the time-randomised configuration is around 10% for both the average execution time and Maximum Observed Execution Time (MOET). For one application (railway) the difference is higher, 40% for the average execution time and 50% for MOET respectively.

It is important to note that the above overhead does not come exclusively from the time-randomised cache hierarchy, but also from the deterministic upperbounding of the floating point unit as introduced in Chapter 4 in order to guarantee the same conditions at analysis and operation. Therefore, the lower execution times of the deterministic platform need to be upperbounded with some (unknown) margin to account for the possible different execution conditions during operation. As a consequence, this not only results in a smaller actual difference between the performance of the two platforms, but also shows that the measurements obtained on the time-randomised configuration can be used with higher confidence.

Finally, the time-randomised platform allows the easy computation of pWCET estimations with a second level of cache, which contributes significantly in the reduction of both average and worst-case execution time as shown in the previous Section. However, in current critical systems, even when the used processors feature a second level cache, as it is the case for the MPC755 in avionics [Wartel *et al.* (2015)], it is disabled since its effect on the WCET cannot be safely predicted with existing methods, sacrificing a significant amount of performance.

6.8 Related Work

The Background on Timing analysis and Cache memories in real-time systems has been presented in Chapter 2. However, in this section we present related work to multi-level caches and complex memory hierarchies. To the best of our knowledge, multi-level cache hierarchies are deemed as hard to analyse and few works have considered them [Hardy & Puaut (2008)][Lesage *et al.* (2009)]. In [Hardy & Puaut (2008)] authors focus on instruction memory accesses on a 2-level non-unified deterministic cache architecture, while in [Lesage *et al.* (2009)] authors focus on data memory accesses on a non-unified cache hierarchy.

One commonality of all the approaches above is that they work on deterministic caches. One of the main characteristics of deterministic caches is that the particular addresses in which objects (i.e. code and data) are located plays a key role in cache performance. This makes that static cache analysis techniques have to deal with an increasingly complex challenge, namely, determining the run time addresses of each access, in addition to having an accurate model of the underlying hardware, i.e. the cache in our case [Wilhelm *et al.* (2008)]. While such information can be obtained for relatively simple programs, deriving runtime addresses, in particular for data accesses, can be regarded as unattainable for industrial-size applications [Mezzetti & Vardanega (2011b)]. Furthermore, hardware efficiency imposes some constraints on cache design, such as using unified second-level caches for data and instructions, considering different inclusion policies, and dealing with write-through and write-back caches as well as write-allocate and non-write-allocate caches.

6.9 Summary

The increasing demand for performance in Critical Real-Time Embedded Systems (CRTES) pushes for the adoption of high-performance features such as multi-level cache hierarchies. However, deriving trustworthy and tight execution time upper-bounds in the presence of such features is deemed as expensive – if at all possible.–

Therefore, there is a need for low-cost industrial-viable means to determine trustworthy and tight Worst-Case Execution Time (WCET) estimates in the presence of multi-level caches.

The advent of Measurement-Based Probabilistic Timing Analysis (MBPTA) together with time-randomised caches has enabled the use of single-level cache memories in an industrial context at low cost. In this chapter, we prove that multi-level time-randomised caches are also MBPTA-compliant by showing that the probabilities of the different events exist. In particular, and for the first time, we enable the use of unified data and instruction second-level caches, implementing different inclusion, cache-write and write-allocation policies without impacting the cost of the WCET estimation. Our results show that 55% average pWCET reductions can be achieved by enabling the use of multi-level caches for CRTES, which obviously decreases the hardware required to schedule critical tasks in CRTES.

Chapter 7

Dynamic Software Randomisation

7.1 Introduction

In the previous chapters we have discussed hardware solutions to enable MBPTA. We have shown that those solutions simplify significantly the computation of the pWCET and allow to efficiently analyse complex hardware such as memory hierarchies featuring several levels of cache, including unified caches which are possible to be analysed for the first time. Moreover, those benefits are combined with low impact on average performance, addressing the performance needs of future CRTES in terms of both average and worst-case performance.

As explained in the external results of the previous chapters, recently, the hardware proposals of this thesis have reached a particularly high Technology Readiness Level (TRL) within the PROXIMA project. A hardware implementation of a LEON3 MBPTA-compatible processor design at RTL-level has been validated and evaluated on an FPGA level, and it is currently available for licensing by Cobham Gaisler [Gaisler (2016)]. Moreover, the MBPTA-compatible features of this processor can be enabled and disabled on will, leaving the choice to the user to decide whether it is going to be used on an MBPTA-compatible mode or on a deterministic setup.

Despite the success of the hardware solutions, the adoption of the proposed hardware modifications by other CRTES semiconductor design firms, which may target larger markets (e.g. automotive) or offer alternative processors in the same markets, may take several years. In addition, the CRTES domain – especially avionics – is traditionally fairly conservative, choosing typically older proven-by-use hardware, whose properties are well understood by its experts. As a result, this may increase significantly the adoption horizon of MBPTA. Moreover, new hardware solutions cannot be used on legacy systems based on COTS processors.

In order to cover this gap and facilitate the adoption of MBPTA by the CRTES industry, we propose equivalent software solutions, that can enable MBPTA on existing hardware. To that end,

in this chapter we extend the applicability of MBPTA to conventional hardware, including direct-mapped and set-associative caches with deterministic placement and replacement policies such as modulo and LRU respectively. We show that the use of randomising compiler techniques that place object code and data in random locations in memory suffices to provide both MBPTA requirements. In particular, random software placement causes the execution time of the software to become random between different executions, with i.i.d. properties preserved. Moreover, it allows each instruction to have an ETP, similar to the hardware solutions, which is the same at both analysis and operation phases of MBPTA, complying with the second MBPTA requirement. Finally, we demonstrate empirically that the randomisation comes at an affordable cost, thus making MBPTA practical for the first time on conventional hardware.

7.2 Compiler and Runtime Support for MBPTA

A *memory object* refers to a memory entity, normally stored in consecutive memory addresses (e.g., functions, basic blocks, data structures), which is manipulated by a software component. These objects can be created off-line by the compiler and the linker, or on-line by the program loader and runtime memory-related libraries.

We define a *cache layout* as the result of mapping all memory objects that form a program into the N cache sets of the cache. Under each cache layout of a program, memory objects conflict in a different manner in cache, which, in combination with the replacement policy, may potentially result in different execution times for the program.

Given a set of memory objects and a fixed sequence of memory accesses, deterministic cache designs generate a single cache layout due to deterministic placement, mapping objects into the exact same cache sets on every execution, and the same sequence of accesses in each cache set due to deterministic replacement. As a result, the execution time does not vary across program invocations[1] as long as *(i) objects are always placed in the same memory location and (ii) the same input data set is used, under which a single path in the program is exercised.*

Therefore, forcing randomised timing behaviour on conventional caches, requires the assistance from a specialised compiler and runtime system that randomises the location of objects in memory, and so the cache layout, before execution begins. For the sake of clarity, we first assume that caches are direct-mapped with modulo placement, so there is no replacement policy. We next generalise our approach by considering set-associative caches implementing a replacement policy.

[1]We consider that other activities, e.g. OS noise, are not considered at WCET analysis but at system integration.

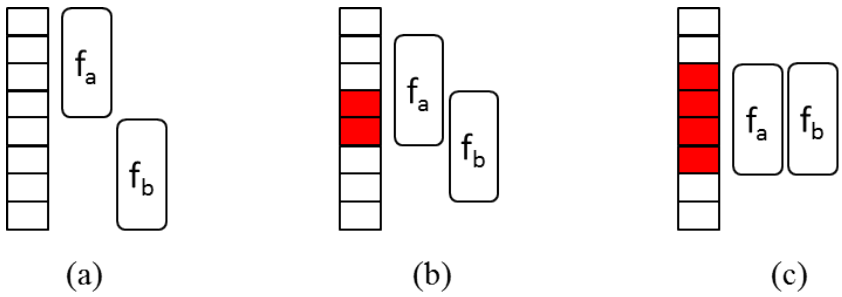


Figure 7.1: Different cache locations of functions f_a and f_b in a direct-mapped cache implementing a modulo placement policy. Red (shaded) locations correspond to cache conflicts among the two functions.

7.2.1 Random Location of Memory Objects

The location of memory objects in random memory positions has the effect of leading deterministic direct-mapped caches to behave similar to random ones. The reason is that randomised layouts lead the cache set to be randomly selected at every new memory allocation, mimicking the behaviour of a random placement policy and so generating random cache layouts across program invocations.

Consider a program formed by a loop in which two leaf functions are called: f_a and f_b, each composed of sequential code. Assume that we execute this program on a processor with a direct-mapped cache implementing a modulo placement policy, and that the total size of the two functions is smaller than the cache size.

Figure 7.1 shows three different possible cache layouts. In Figure 7.1(a) the two functions are placed in consecutive memory positions that do not collide with each other, thereby having no cache conflicts among objects (*inter-object conflict*). However, if they are placed in memory positions such that the modulo function makes two pairs of addresses from the two functions collide into the same cache set, the effectiveness of the cache will be decreased because of inter-object conflicts, as shown in Figures 7.1(b) and 7.1(c). Randomly mapping memory objects results in random cache layouts, each leading to potentially different execution times.

Note, however, that cache conflicts within memory objects (*intra-object conflicts*) are deterministic. For instance, if the size of f_a size exceeds the size of the cache, some of its cache lines would be mapped into the same cache set and would conflict. MBPTA requires execution times collected to capture the behaviour of the program under analysis. Such behaviour can manifest in only two ways: (i) constant or (ii) probabilistic, because deterministic non-constant behaviour cannot be modelled with probabilities. If memory objects are defined at a granularity (e.g., function code, stack data) so that their internal layout cannot change, *all* runs of the program will have identical intra-object conflicts and so variation on execution time will be only produced due to random placement of objects in memory.

7.2.2 Formal Justification for Applicability of MBPTA

MBPTA requires the existence of an ETP for each instruction [Cucu-Grosjean *et al.* (2012)]. We now argue why randomised layouts guarantee the existence of an ETP for each instruction i accessing to a given cache line, i.e. $ETP(i) = \{l_{hit}, l_{miss}\}\{1 - P_{miss_i}, P_{miss_i}\}$.

A memory operation i accessing cache line c_a belonging to object a will conflict in the cache if there exists another cache line c_b belonging to another object b that is mapped into the same cache set. A modulo placement policy uses some index bits of the memory address to identify the cache set. This approach logically divides the address space into $\frac{M}{N}$ different chunks, where M is the total memory size divided by the cache line size. Within each chunk, memory addresses are mapped to the N cache sets in the same manner. Therefore, the memory chunk in which a memory object is placed is not relevant, but the offset within the chunk. Thus, if we randomly place those objects with respect to memory chunk boundaries (either in a new chunk or in an already in-use chunk if objects do not overlap), inter-object conflicts will occur randomly, and each object will have exactly $\frac{1}{N}$ different placements with respect to the cache (memory objects must be aligned to cache line boundaries, which is usually the case).

Hence, assuming an arbitrary sequence of memory accesses to cache lines $c_a, c_{b_1}, c_{b_2}, \cdots c_{b_m}, c_a$ belonging to objects $a, b_1, b_2, \cdots b_m, a$ respectively, the probability that the second access to c_a is a miss is $P_{miss}(i) = 1 - \left(\frac{N-1}{N}\right)^m$, where $\frac{N-1}{N}$ is the probability that a particular cache line is not placed into a particular cache set and m is the number of unique cache lines accessed in between the two accesses to the cache line c_a.

As a consequence, each execution of the program results in a potentially different cache miss count since different execution times occur with different probabilities, as a consequence of the fact that program instructions have ETP describing their timing. Thus, this results in a random total execution time. Moreover, since each program execution is independent of the other executions, and the placement of the objects in memory is random, the resulting execution time has i.i.d. properties. Provided that software randomisation is not only used at analysis time, but also during operation, the ETP of each instruction exists at both MBPTA phases. In addition, those ETPs have exactly the same values across those phases, hence satisfying the second MBPTA requirement, that requires events at analysis to upperbound or match – in a probabilistic manner in the particular case – the events occurring during operation. Therefore, software randomisation satisfies both MBPTA requirements.

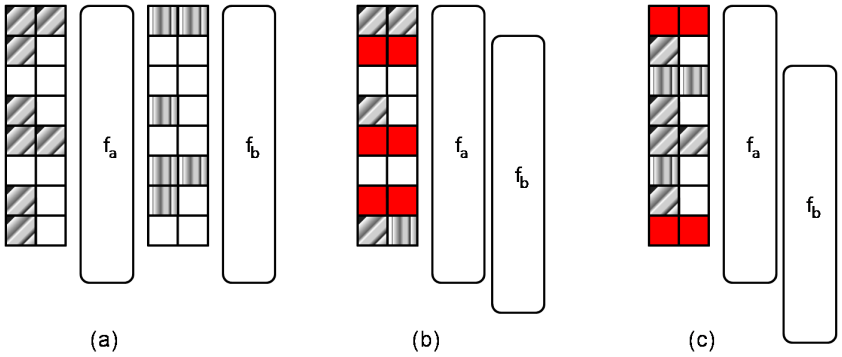


Figure 7.2: Cache locations and layouts of functions f_a and f_b in a deterministic two-way set-associative cache. Red regions denote the cache way conflicts between the two functions.

7.2.3 Effect of Replacement Policy

A cache with a deterministic replacement policy can be made to behave as if it was using random replacement by randomising the order of memory accesses to each particular cache set. Random layout changes the mapping of objects to sets on each execution, thus randomising the order of accesses to each cache set in a random (and thus probabilistic) way.

This effect is illustrated in the following example. Figure 7.2(a) shows the cache layout of placing f_a (left) and f_b (right) into a two-way set-associative cache. None of the functions has a sequential structure and so they allocate two lines in some cache sets, and only one or zero lines in other sets. This example reflects the cache utilisation of the dynamic invocation of functions when some parts of the code can be skipped due to jump instructions.

When the two functions are co-located in the same cache (Figures 7.2(b) and (c)), cache lines belonging to f_a and f_b may conflict in some cache sets. Such conflicts will depend on where functions have been randomly placed. Thus, if functions are located as shown in (b), there will be conflicts in 3 cache sets (marked in red), as 3 or 4 different cache lines are candidates for only two ways. This is not the case for the last cache set, in which cache lines belonging to f_a and f_b fit. Instead, if functions are located as shown in (c), there will be conflicts in only 2 cache sets (marked in red), different from the ones that occur in (b).

As shown, random layout of memory objects randomises the cache lines from each object colliding into each set, so the accesses to each cache set (those determining the behaviour of deterministic replacement policies such as LRU) will be determined by random events (the particular random layout). This ensures that inter-object conflicts do not occur deterministically, and their effects can be captured by ETPs.

Note that using a hybrid solution, combining randomised layout with hardware random replacement, would also cause both inter- and intra-object conflicts to occur probabilistically but would increase the degree of randomisation [Cobham Gaisler (2011)] [ARM (2006)].

7.2.4 Randomising Compiler and Runtime System

Dynamic Software randomisation performs the randomisation of memory objects dynamically at runtime, using *Stabilizer* [Curtsinger & Berger (2013)]. This tool is composed of a combination of a compiler pass (using LLVM [Lattner & Adve (2004)]) that modifies appropriately the application's code and a runtime system that is in charge of performing the relocation of objects in memory. Stabilizer uses the DieHard memory allocator [Berger & Zorn (2006)] as the basis of its runtime system to perform efficient ($O(1)$) dynamic layout randomisation. The entire process is applied in user space, however some support from the operating system is required.

The memory objects whose location can be randomised are code (functions), stack frames, global data and dynamically allocated objects from heap.

In code randomisation, the compiler pass keeps track of the functions that are going to be relocated during the program's execution. These functions are visited during program start-up and a trap generating instruction is placed in their beginning. Whenever a function is visited, the generated trap is assigned to a signal handler which performs the following actions: allocates memory in a random location for the function, copies the code of the function and replaces the first instruction of the original location of the function with a jump to the new function location. Therefore all new calls to the function are redirect to the new random location.

Stack randomisation is implemented by making the stack non-contiguous. The compiler pass inserts code in each function's prolog and epilog, which adjusts the stack frame with an offset (ranging between 0 and the way size of the largest cache of the memory hierarchy) read from a table assigned to this function. This table is initialised in the program start-up with a random positive value, which randomises the location of the stack frame's initial address.

Global data are randomised at program's start up, using redirection tables.

The random placement of dynamically allocated objects from the heap is implemented using a random memory allocator (Die-Hard). The same allocator is used for the all the dynamic allocations used by runtime (code, global data).

In all cases, the random offset introduce among objects, i.e. functions, stack, global data and dynamic memory, ranges between 0 and at least the way size of the largest cache of the memory hierarchy.

7.2.5 Detailed Implementation Description

Code randomisation: The randomisation of functions is done by relocating each function's code into a random memory location at program startup time. A *Relocation Table* (RT) is placed at the end of each new relocated function to identify the addresses of all globals and functions accessed by the relocated function (see Figure 7.3). Stabilizer's [Curtsinger & Berger (2013)] compiler transformation rewrites all accesses to globals and other functions to indirect accesses through the RT. Figure 7.3 shows an example of two functions f_a and f_b with the latter calling the former (dotted line). The Figure also shows the layout of the functions once reallocated. Function relocation is carried out in two phases:

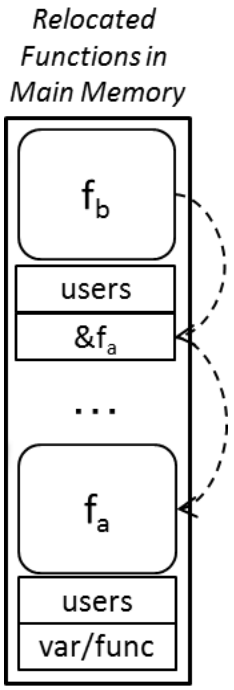


Figure 7.3: Randomisation of code frames of functions f_a and f_b into the main memory.

- *Initialisation-Relocation.* At program startup, the Stabilizer runtime library allocates, for each function and in a random location, a sufficiently large block of memory from the DieHard [Berger & Zorn (2006)] allocator and copies its code to this location. The runtime then generates a RT immediately after the new function location, with all entries pointing to the original locations of any called functions and globals.

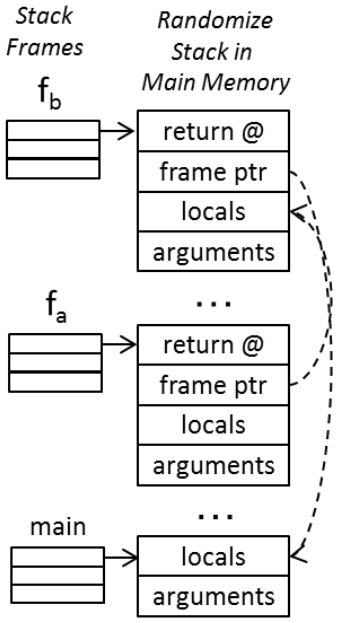


Figure 7.4: Randomisation of stack frames of functions f_a and f_b into the main memory.

- *Redirection.* Then, Stabilizer overwrites the first instruction of the original function location with a static jump to the new location. This forwards all future calls to the random function location.

Stack Randomisation: Stabilizer randomises the stack by making it non-contiguous. Stabilizer allocates a block of memory *for every <function,nesting level> pair*, with the allocation being done at function call time. Hence, if under the main function we have the sequence of calls $f_a(); f_b(); f_a();$, on the first call to $f_a()$ stabilizer allocates a block, which is not deallocated and reused for the second call to $f_a()$. If a function is called nestedly (e.g., $f_a()$ calls $f_a()$), Stabilizer allocates a new block of memory for every recursion level.

Every function has a depth counter and frame table that maps the depth to the corresponding stack frame (see Figure 7.4). The depth counter is incremented at the start of the function and decremented just before returning. On every call, the function loads its stack frame address from the frame table. If the frame address is null, the Stabilizer runtime allocates a new frame.

Global/static Variables Randomisation: Globals[1] location can be randomised analogously to function code at program startup and access to globals is then performed through indirections and pointers.

[1]We refer to global and static variables as *globals* for the sake of convenience.

7.3 Results

7.3.1 Experimental Setup

All measurements presented here are conducted on the simulation environment described in Chapter 3, modelling a single level memory hierarchy composed of separate instruction and data caches and main memory. Both caches model a conventional 4KB set-associative cache with 8 ways, 32 sets and 16-byte line size, implementing a modulo placement policy with LRU or random replacement policy.

The data cache implements a write-through, no-allocate write policy. The only source of execution time variation in the processor is the cache, which in both instruction and data has a hit latency of 1 cycle and miss latency of 100.

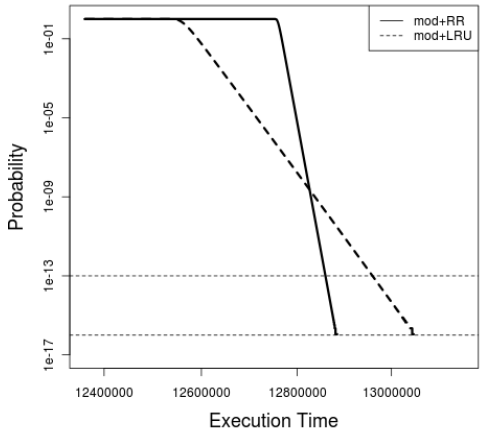
In this chapter we selected to present only a subset of the EEMBC Autobench benchmark suite [Poovey (2007)] for evaluation: *a2time01*, *cacheb01* and *puwmod01*, in order to demonstrate the different types of behaviour that software can experience in term of pWCET distributions. In later Chapter (Section 9.3 and Figures 9.4 and 9.5 we present detailed results for all benchmarks, using a cut-off probability of 10^{-15} per run for comparison with the proposed software randomisation variant of that chapter). To compute pWCET estimates, we use the method in [Cucu-Grosjean *et al.* (2012)].

7.3.2 Independence and Identical Distribution Tests

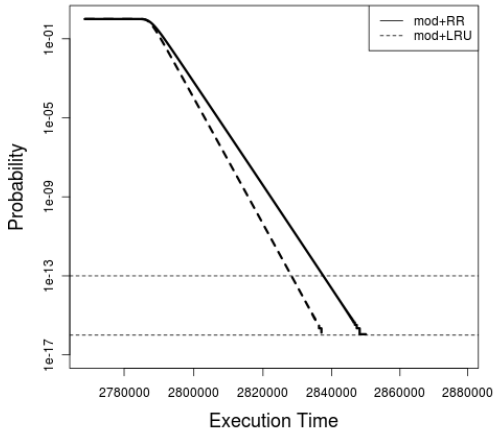
In order to test independence and identical distribution, we use the Wald-Wolfowitz independence test [Bradley (1968)] and the two-sample Kolmogorov-Smirnov identical distribution test [Boslaugh & Watters (2008)] as described in [Cucu-Grosjean *et al.* (2012)]. We have evaluated i.i.d. properties for the three benchmarks under analysis considering two cache configurations implementing modulo placement and LRU replacement policies (labelled as *mod+lru*) and modulo placement and random replacement policies (labelled as *mod+rr*). For all cases, the p-values obtained (not shown due to space constrains) pass the tests ($p - value > 0.05$ for identical distribution and $p - value < 1.96$ for independence), indicating that both cache configurations provide i.i.d. execution times when we randomise function and stack layout.

7.3.3 pWCET Estimates

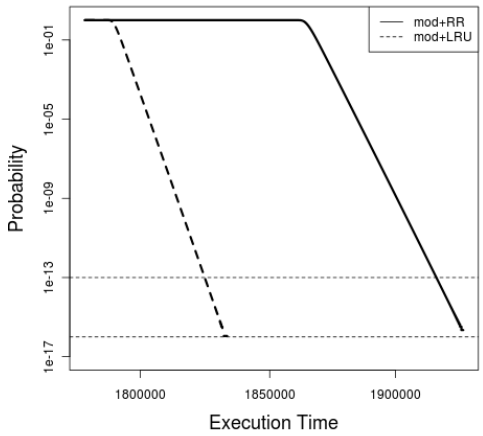
Figure 7.5 shows the pWCET estimates obtained with MBPTA [Cucu-Grosjean *et al.* (2012)] for *a2time* (a), *cacheb* (b) and *puwmod* (c), considering our two cache configurations. In all cases, we require less than 1,000 runs to project the tail.



(a) a2time01



(b) cacheb01



(c) puwmod01

Figure 7.5: pWCET estimations of caches implementing modulo + LRU and modulo + random replacement (labelled as *mod+lru* and *mod+rr* respectively).

The effect of using a random replacement policy instead of LRU replacement policy depends on the program. If we consider the pWCET estimates at an exceedance probability of 10^{-15}, random replacement increases the pWCET estimate of *puwmod* by 5% over LRU. However, for *a2time*, random replacement reduces the pWCET estimate by 2% over LRU. For *cacheb*, there is almost no variation in pWCET estimates between random and LRU replacement policies (less than 1%).

These results support the analysis of Section 7.2: software randomisation makes it possible to apply MBPTA without requiring additional hardware support such as a random replacement policy. Nonetheless, the use of a random replacement policy remains desirable as it further randomises inter-object and intra-object conflicts.

7.3.4 Overhead

Our software randomisation approach introduces some overhead due to the relocation of functions and stacks. The former copies each function to a new location. The latter causes each function call to move the stack to a new location.

In order to understand the impact on pWCET estimates, we repeat the same experiment as in the previous section but on top of a FA-RR cache, where software randomisation has no effect on timing behaviour. As a result, the pWCET estimate increment observed with respect to not applying software randomisation is only due to the overhead. We have designed a specific synthetic benchmark consisting of a loop which contains calls to four distinct functions. This structure is very similar to EEMBC.

When considering the pWCET estimate increment at an exceedance probability of 10^{-16} of the synthetic benchmark when applying both function and stack random memory location, we observe that, as we increase the number of iterations, the compiler overhead is reduced, as the relative impact of the initialisation part is reduced. Executing only 100 iterations, the software approach increases pWCET estimates by almost 10x. Such an increment is reduced to 2x when executing 1,000 iterations and only 66% when executing 10,000 iterations.

7.4 External Results

In addition to the results obtained in the scope of this thesis and presented so far in this Chapter, we have also evaluated the proposed dynamic randomisation technique in the the context of the FP7 projects PROARTIS and PROXIMA in collaboration with industrial partners.

In particular in [Wartel *et al.* (2015)] two avionics case studies have been evaluated on top of the simulation environment we have developed in this thesis and described in Chapter 3, implementing a conventional PowerPC 750 processor. This work demonstrates the applicability of our dynamic software randomisation solution in large scale applications deployed in IMA systems.

According to the obtained results, our method is able to provide the i.i.d. properties which are required for the MBPTA applicability and the estimated pWCET is tighter than the ones obtained with the current industrial practice of 20% margin on top of the maximum observed execution time (MOET). In fact the pWCET is found to be only 12.2% higher than the MOET. Moreover, the average

performance degradation with respect to the non-software randomised application variant is between 0.7% and 12% for each application respectively. Finally, this work performs a comparison between hardware and software randomisation on these avionics application. The overhead of software randomisation with respect to the hardware randomisation is only 10.9% higher. This way we show that although hardware solutions are more efficient, software randomisation is competitive and less intrusive as the hardware can remain unchanged.

In the context of the PROXIMA project, our dynamic software randomisation technique has reached a high Technology Readiness Level (TRL) by being ported and evaluated on several COTS platforms and industrial real-time operating systems. In particular the dynamic software randomisation has been ported to the SPARC v8 ISA, supported by the LEON3 processor as well as to the PowerPC-based P4080. Moreover it has been integrated with numerous personalities of a commercial real-time operating system developed by SYSGO (PikeOS native, PikeOS RTEMS, PikeOS ARINC 653 and RTEMS-SMP) on both hardware platforms.

For the evaluation, four industrial case studies have been executed on the LEON3 platform, the same ones used for the external results provided for the hardware solutions in Section 6.7, while on the P4080 only the two avionics case studies – the same ones with [Wartel *et al.* (2015)] – have been evaluated [Agirre *et al.* (2016)]. The results from the evaluation of the aerospace case study have been also published in [Kosmidis *et al.* (2017)].

Apart from the particular microarchitecture (pipeline stages, in-order/out-of-order, branch prediction, instruction and data cache sizes, etc), the only difference between the architecture considered in the previous section and those hardware platforms is the presence of a second level unified cache. Similarly to the case of the multilevel hardware designs introduced in Chapter 6, software randomisation works also in the presence of multiple levels of cache, providing ETPs for each instruction. From the implementation perspective, software randomisation needs to consider random placement in chunks equal to the larger cache way of the available caches, which is the last level cache in both considered platforms.

In general, results on both COTS platforms are in line with the evaluation presented in this Chapter [Agirre *et al.* (2016)]. In all the cases software randomisation was able to provide identically distributed execution times, but in few cases the independence test has been failed. This was the case for some small tasks (functions) under analysis which fit perfectly in the cache, so that software randomisation was not able to create enough variability in the small number of the different placements experienced in cache. In terms of performance, the obtained pWCET estimates with MBPTA were lower than the current industrial practice of adding a 20% margin over MOET or very close to it.

Regarding the comparison of the FPGA hardware randomised LEON3 versus the software randomisation on the FPGA COTS LEON3, the measurements confirm the results obtained with our simulation platform [Wartel *et al.* (2015)]. In particular, software randomisation has a higher overhead compared to the hardware solutions, but it is within an affordable range. Interestingly, in the space case study, software randomisation provided higher average performance than the deterministic execution, due to an 1% increase in the cache hit ratio. This was a result of the default memory layout of the application, which happened to be problematic, and software randomisation allowed to explore better cache layouts yielding higher performance despite the additional redirections used at runtime.

Despite the small overhead introduced by software randomisation, current CRTES already experience significant performance loss from disabling second level caches and the addition of engineering margins over MOET. However, software randomisation and the use of MBPTA which it enables, allow CRTES to get higher performance out of existing COTS hardware, while providing high confidence over the obtained pWCET estimations.

To sum up, software randomisation has been proven as a good alternative when hardware solutions cannot be applied and can be a significant MBPTA enabler to industrial systems, until MBPTA-compliant hardware is available.

7.5 Related Work

Dynamic software randomisation has found several applications in the literature. Bhatkar et al. [Bhatkar *et al.* (2005)] introduce stack randomisation as a method for thwarting stack-smashing based security exploits. Berger and Zorn's DieHard system [Berger & Zorn (2006)] randomises the layout of objects on the heap to provide probabilistic memory safety, tolerating memory management errors. Mytkowicz et al. [Mytkowicz *et al.* (2009)] show that the memory layout may degrade a program's performance by as much as 300%, and propose a random function layout in memory, varying the link and the size of environmental variables. Curtsinger and Berger [Curtsinger & Berger (2013)] propose dynamic software randomisation as a means of performance evaluation, in order to isolate the effect of statistically relevant data from platform noise, showing that the performance difference between -O2 and -O3 optimisation levels in the LLVM is indistinguishable from noise.

Despite the popularity of software randomisation in the fields of security and high performance, it has not found any applicability in CRTES, where WCET of the program must be derived and deterministic behaviour has traditionally been considered the ideal. Therefore, our proposal for using software randomisation in order to enable the use of MBPTA on COTS hardware is the first work of this kind in the CRTES literature.

7.6 Summary

This chapter presents an approach that extends the applicability of MBPTA to conventional cache designs, e.g. implementing modulo placement and both LRU and random replacement policies, via a software-only randomising compiler and runtime system. Placing functions and stack frames in random memory locations causes deterministic modulo placement policies to exhibit the same behaviour as a random placement policy, yielding observed execution times that satisfy both properties required by MBPTA: a) they are independent and identically distributed (i.i.d.) and b) they follow the same probabilistic behaviour at both analysis and operation. We provide a formal argument explaining how software randomisation enables the derivation of execution time profiles (ETPs) for each memory operation. Finally, we empirically show that software-only randomisation causes deterministic caches to behave as if they were random, making it possible to use MBPTA on top of conventional hardware.

Chapter 8

Static Software Randomisation at Compiler/Linker Level

8.1 Introduction

In the previous chapter, we proposed dynamic software randomisation as a manner to enable MBPTA to be used on conventional hardware. We have shown that this technique has reached a significantly high TRL inside the PROXIMA project, and that it has been evaluated in industrial setups. However, the dynamic software randomisation cannot be applied on certain architectures that prevent self-modifying code, such as the ones used in the automotive domain. Moreover, the presence of self-modifying code and pointer redirections, require additional effort for the verification and validation (V&V) in particular CRTES domains, which can potentially increase the cost of adopting those solutions.

In order to cover this gap, in this chapter we propose a new approach to software randomisation in which randomisation is performed *statically*. While dynamic software randomisation (DSR) relies on including some randomisation code in the program executable so that, every time the program is invoked, memory objects are randomly placed, static software randomisation (SSR) relies on generating several binaries for the same program. In each binary memory objects are shifted appropriately to produce the same effect as if those objects are placed at random, yet statically determined, locations at runtime. By using SSR, modifications on the binary introduce neither indirections nor extra pointers. As a consequence, functional verification remains no more complex than without software randomisation, requiring a qualified compiler that generates the SSR binaries. Analogously to DSR, SSR allows computing the probability that a given arrangement of objects in memory leads to a timing violation.

8.2 Static Software Randomisation

This section introduces how DSR challenges functional verification, and how this problem is addressed by our SSR techniques. We also review the implications of performing SSR.

8.2.1 Functional Verification of Software

Programs implementing safety-related functions need to meet ISO26262 standard [International Organization for Standardization (2009)] requirements in the automotive domain. Rigor is needed to meet standard requirements, and thus, a number of software design principles are listed in those standards to facilitate a successful software functional verification. Among those principles, in the ISO26262 we can find the following ones: (i) a limited use of pointers, (ii) recommendations against the use of dynamic objects, and (iii) no hidden data flow or control flow. This is particularly true for the highest safety levels (e.g. ASIL-C and ASIL-D).

DSR challenges functional verification of software in different ways. First, indirections for functions, stack frames and global/static variables occur through pointers. Second, functions (code), globals and stack frames use dynamic objects allocated by the DieHard [Berger & Zorn (2006)] allocator. And third, data and control flow occurs through pointers, thus making it non-obvious.

As a consequence, generating use cases for software testing is challenging as the analysis of boundary values and error guessing, as listed in ISO26262, is hard when pointers and dynamic objects are used. Furthermore, the fact that functions are copied dynamically in new memory locations, thus writing code in data memory segments for its later execution, makes programs have self-modifying code, which some architectures may not support as the memory management unit may prevent memory pages in the data segments from being fetched for their execution.

8.2.2 Static Code Placement Randomisation (SSR-code)

DSR-code relies on placing functions (code) at random locations every time the program is run. For that purpose, functions are compacted in the binary (no empty space between functions) and they are copied to the desired random locations when the program is run. Such an approach increases the memory space needed in the data segments at runtime to copy the code, but the size of the binary is increased negligibly to include the function copy code and indirect function calls.

In the case of our new *SSR-code*, random locations for functions are determined statically at compile time and such locations are already reflected in the function layout in the binary. This could be done by simply introducing some random shift

(1) **Input:** F function list
(2) WS = cache way size (bytes)
(3) LS = cache line size (bytes)
(4) $F_{place} = \emptyset$
(5) **for** $i = 1$ **until** $cardinality(F)$ **do**
(6) $\quad pad = \left(random()\ mod\ \frac{WS}{LS}\right) \cdot LS$
(7) $\quad F_{place} = F_{place} \bigcup \{F, sizeof(F), pad\}$
(8) **end for**
(9) $Align = 0$
(10) $Binary$ = empty file
(11) **While** $F_{place} \neq \emptyset$ **do**
(12) $\quad MinPad = WS$
(13) $\quad$ **for** $i = 1$ **until** $cardinality(F_{place})$ **do**
(14) $\quad\quad Waste = (F_{place}(i) \rightarrow pad + WS - Align)\ \ mod\ WS$
(15) $\quad\quad$ **if** $Waste \leq MinPad$ **then**
(16) $\quad\quad\quad Best = F_{place}(i)$
(17) $\quad\quad\quad MinPad = Waste$
(18) $\quad\quad$ **end if**
(19) $\quad$ **end for**
(20) $\quad Binary = append(Binary, MinPad, Best)$
(21) $\quad Align = (Best \rightarrow pad + Best \rightarrow size)\ \ mod\ WS$
(22) $\quad F_{place} = F_{place} - Best$
(23) **end while**
(24) **Return:** $Binary$

Figure 8.1: Algorithm to randomly place functions in the binary.

(padding) between functions; however, this would increase binary size inordinately. In order to achieve the same effect efficiently, a number of steps must be taken to place functions randomly while minimising the size of the binary. Those steps are described in the algorithm in Figure 8.1.

We assume a cache deploying modulo placement as it is one of the most common placement functions. Given a cache of CS bytes with W ways, the size of a way is given by $WS = \frac{CS}{W}$. Given a function whose initial (instruction) address is $@_A$, its first instruction will be placed in cache set $S_A = \left\lceil \frac{@_A \bmod WS}{LS} \right\rceil$ where LS is the line size in bytes. Note that, if we shift the initial address by WS, $@_A + WS$, the first instruction will still be placed in S_A. Further, no assumption is made regarding the cache line alignment of instructions. Their original alignment with respect to the cache line is preserved when introducing a random shift if its size is a multiple of the cache line size.

Given the function list (F), we traverse it computing a random offset (or padding) for each function that determines its alignment w.r.t. the beginning of the cache way (lines 5-8). This padding (*pad* in the figure) is a multiple of the cache line size[1]. F_{place} is a tuple containing the function, its size and the padding assigned to that function.

Once each function has its own padding (offset), any given function k_i can be described as a contiguous memory area whose first instruction is mapped to set $S_{k_i}^{init}$ and whose last instruction is mapped to set $S_{k_i}^{last}$. The algorithm then looks for the function k_0 whose $S_{k_0}^{init}$ is the closest to 0 (or even 0). Once this function is placed in the binary, the algorithm looks for a function, from the list of not-yet-allocated functions, k_1, whose initial set is as close as possible to the set in which the last instruction of k_0 was assigned (lines 13-19). That is, the algorithm reduces the wasted space between $S_{k_i}^{last}$ and $S_{k_{i+1}}^{init}$ (lines 15-18). Note that the problem of minimising the total padding in-between functions is a NP-complete problem. In the algorithm, once each function has its own offset with respect to the first set of a cache way, we initialise the next alignment available (line 9) that can be used to place a function and the binary (line 10). Then, we traverse the set of functions, which includes their respective sizes and paddings (line 11). We initialise the minimum padding found to the largest value possible in line 12 (the size of a cache way). For each function in the set (line 13) we compute how much space would be wasted in between this function and the last one placed in the binary if we placed the current function next (line 14). If such wasted space is equal or lower than the smallest one found (line 15), then we pick the current function as the best candidate to be placed next (line 16) and update the minimum padding found (line 17). Once all functions have been examined and the best candidate found, we append such function to the binary adding the corresponding padding (line 20), update the alignment desired for the next function to be placed (line 21) and remove the function placed from the set (line 22). This process repeats until all functions are placed and the layout of all functions in the binary is obtained. The algorithm we propose for *SSR-code* is a greedy algorithm whose cost is $O(N^2)$ w.r.t. the number of functions.

Once the functions are conveniently arranged in the binary, their initial addresses are fully known and can be used for calling them, as in a non-randomised program. Therefore, indirections needed for *DSR-code* are no longer needed. Moreover, since functions do not have to be written into the data memory space, as it is the case for *DSR-code*, which relies on self-modifying code, memory protection is not challenged. Instead, instructions are fetched only from the code segment.

[1]We assume that functions are cache line aligned as this improves spatial locality, although *SSR-code* is not limited to this assumption.

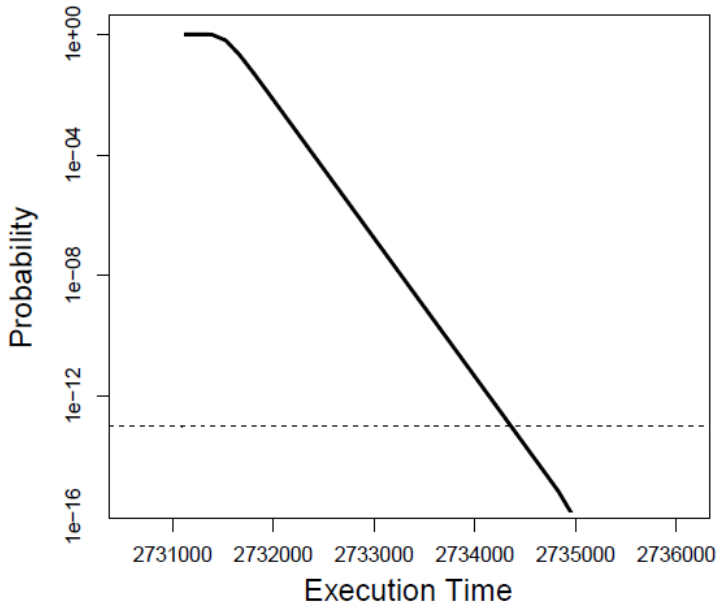


Figure 8.2: pWCET distribution in processor cycles for an industrial program.

8.2.3 Static Stack Frame Randomisation (SSR-stack)

While functions exist in the binary, stack frames are created dynamically. In a non-randomised binary, the stack frame for a new function call is created right after the last stack frame. *DSR-stack*, instead, allocates stack frames in random locations and uses them to break such determinism. However, the location of those stack frames is unknown statically.

To make the location of randomly allocated stack frames known at compile time, we propose *SSR-stack* which, just before allocating a new stack frame, adds a random padding in the stack frame, computed at compile time. The added padding, as for the case of functions, ranges between 0 and the size of a cache way (WS) since address placement in cache wraps up beyond the cache way size. By doing so, stack frame placement is random and different across functions with *SSR-stack* as it is the case for *DSR-stack*. Note that, unlike *DSR-stack*, *SSR-stack* imposes that nested calls to the same function use the same (random) padding. Although this decreases the degree of randomisation (nested calls use the same padding value), padding is still random across different images (binaries) so i.i.d. properties across images still hold as needed by MBPTA.

SSR-stack adds to each function an instruction that decreases the stack pointer by padding bytes. Since the value of padding is known at compile time, no extra pointers are used and all addresses can be determined as in a non-randomised binary, thus not increasing functional verification complexity.

8.2.4 Static Global/Static Variable Randomisation (SSR-globals)

As indicated before, the case of globals is analogous to that of functions. Therefore, the algorithm in Figure 8.1 can also be used to randomise the location of globals. If globals are placed in different segments (i.e. `.data` and `.bss` segments), then the algorithm must be applied individually to each segment. Still, the complexity of the problem is the same as for code since objects characteristics are fully known at compile time and hence, their location in the binary can be randomised analogously.

8.3 Deploying DSR and SSR

In this section we study the impact of deploying DSR and SSR in an automotive computing system, which requires understanding the pWCET estimates obtained with MBPTA. Figure 8.2 shows the pWCET obtained for a representative program [Wartel *et al.* (2013)]. The X-axis shows time, with the scale not starting at 0, and the Y-axis probabilities in logarithmic scale. The function represents the pWCET estimate for the program. We observe that the pWCET function has a steep gradient. This means that the increase in pWCET experienced when decreasing the cutoff exceedance probability is small. For instance, the increase in pWCET from 10^{-12} to 10^{-16} is less than 1%[1].

8.3.1 DSR

With DSR, code and data placement are randomised across executions of the binary of a program, so that the exceedance probabilities of the pWCET estimates obtained for that program apply at the end-to-end run granularity. For instance, an exceedance probability of 10^{-16} implies that an execution of the program can exceed the corresponding pWCET estimate with at most that probability for that execution instance alone. In order not to exceed a timing failure rate per hour (e.g., 10^{-16}), if the program is executed, for instance 10^3 times per hour, the system designer should choose as pWCET estimate the value at exceedance threshold 10^{-19}, so that it is guaranteed probabilistically that the accumulated timing failure rate of all instances of execution of the program in one hour is below 10^{-16}.

[1]To appreciate how small the 10^{-16} cutoff probability is, consider that Extinction Level Events (ELE), such as an asteroid hitting the Earth, are estimated to happen about once every 100 million years, hence at an arrival rate of 10^{-16} per second, or 10^{-12} per hour.

8.3.2 SSR

With SSR, code and data placement are randomised across images (binaries). Thus, i.i.d. properties are attained at *image level* rather than at *end-to-end* run granularity. Thus, timing failures apply at the granularity of binary instead of at end-to-end run granularity. In order to derive pWCET estimates with MBPTA, during the analysis phase, SSR deploys an automated approach in which N experiments are run, each with a different binary. The collected execution time from each run is fed to MBPTA to derive a pWCET estimate. Note that N is in the order of hundreds as shown in previous studies [Cucu-Grosjean *et al.* (2012)[Wartel *et al.* (2013)].

For the system operation (in contrast to analysis, the two phases in MBPTA applications as introduced in Section 4.3.1) there are two different approaches possible for SSR: (i) deploying a different binary in each system or (ii) deploying one of such binaries in *all* systems.

SSR with different binaries per system unit: In this approach the probability of timing failure of the program per system unit deployed is independent across units. If the pWCET of the program is not to be exceeded with a probability higher than 10^{-22} and 10^{6} units are delivered, there is a probability of 10^{-16} of exactly one system in which that program binary experiences timing failures and 10^{-32} that it would happen in two different units. The downside of this approach is that it implies each unit having a different binary[1], which may not be acceptable if each individual unit is not fully tested.

SSR with the same binary: In this approach a single binary is generated with SSR and deployed in all systems. Then, if the binary exceeds the pWCET, it may do so *in all units*. However, this can be made to occur with negligible probability, in the same order of probability of an Extinction Level Event (ELE) to occur. Hence, if for instance, the pWCET is not to be exceeded with a probability higher than 10^{-16} it is much more likely to experience an ELE than a timing failure.

8.4 Evaluation

In this section we evaluate the overheads introduced by SSR. Since SSR is performed statically, there is no need to link any runtime library as it is the case for DSR. Finally, performance overheads have been inferred from the application of DSR on the cycle-accurate execution-driven simulator we developed for this thesis and described in Chapter 3 modeling a conventional memory hierarchy composed of first level separated instruction and data caches (1 cycle hit, 100 cycles miss),

[1] As binaries are randomly generated, it can be the case that different units get identical binaries, although this is highly unlikely to occur.

Table 8.1: Binary size overheads for the case study.

Way size	1KB	2KB	4KB	8KB
Code	4,866,386	4,923,387	5,040,105	5,288,843
% increase	2.8%	4.0%	6.5%	11.7%
Data	673,145	693,159	734,608	815,765
% increase	4.1%	7.2%	13.6%	26.1%
Code+data	5,539,531	5,616,546	5,774,713	6,104,608
% increase	2.9%	4.4%	7.3%	13.4%

and main memory. Both instruction and data caches are 8KB 8-way set-associative with 32B lines. Both caches implement modulo placement and LRU replacement policies. The data cache implements a write-through, no-allocate write policy.

8.4.1 Memory Overheads

SSR-code and *SSR-globals*

SSR-code and *SSR-globals* are the main source of memory overheads when applying SSR. As shown in Section 9.2.2, some space is left between functions (and also between globals) in the binary so that their placement in cache is random. Next we analyse the overheads incurred when applying *SSR-code* to an industrial-size (case-study) application [Wartel *et al.* (2013)][1]. This application consists of around 5,000 functions whose sizes range between few bytes and 300KB. The total size of those functions is 4.7MB if they are enforced to be aligned with cache line boundaries assuming a cache line size of 32B. To analyse the sensitivity of *SSR-code* to cache way size we consider cache way sizes of 1KB, 2KB, 4KB and 8KB. Table 8.1 reports the average size in bytes of the code segment obtained for each way size for 1,000 different static software randomisations of the application together with the relative size increase w.r.t. the original code segment size. Maximum values are relatively close to the average , thus proving the stability of our approach. Also, as the way size increases, inefficiency also increases because the average size of the padding between different functions in the binary also grows.

The same analysis can be applied to globals as for functions in the industrial application. In this case the application consists of around 70,000 globals whose sizes range between few bytes and 24KB. Those globals are all in the `.bss` segment, as it is initialised to zero values. Uninitialised segments such as the `.data` one are not recommended for safety-critical applications.

[1] We have corroborated that the characteristics of this avionics application reasonably resemble some real automotive applications in terms of function count and function size.

The total size of those globals is 2.3MB if they are enforced to be aligned with cache line boundaries assuming a cache line size of 32B and only 640KB otherwise. In order to reduce such inefficiency we have packed globals so that they fill full cache lines and therefore, they can be kept cache line aligned. This leads to 18,000 objects, but most of them fill just one cache line. Small objects are prone to leave many gaps in between in the binary that cannot be filled. In particular, for a 1KB way size we observe a 60% data segment increase with 32B objects. Thus, we have packed globals into larger objects (1KB each) whenever possible for a total of 590 objects by simply packing the largest objects that do not exceed the cache line size. This still leads to around $32^{590} \approx 10^{900}$ different potential object placements for a 1KB way size[1], so the degree of randomness attained is still huge. Note that as discussed in the previous chapter, the degree of randomness has been proven not to affect the correctness of the approach, but the value of the pWCET estimates, if few different object placements can be obtained, which is not our case at all. Note that by packing globals into sizes not exceeding the cache way size, no conflict can occur across those globals and such behaviour holds both at analysis time and at deployment. Results are shown in Table 8.1. Overheads are similar to those for *SSR-code*.

Total results for the binary, including code and data segments are also reported in Table 8.1. As shown, binaries are expected to grow little due to SSR, with cache way sizes in this range.

Sensitivity Analysis: We have also performed a sensitivity study considering random function sizes between 128B and 2048B varying the number of functions between 10 and 1,000. Results are shown in Figure 8.3. As for the case study, inefficiency grows for larger cache ways. Furthermore, we also observe that the larger the number of functions, the lower the relative overhead since it is easier to find functions to be placed with little padding. In particular if cache way size is 1KB the binary size overhead decreases from 19.2% (10 functions only) down to 2.0% (1,000 functions). Conversely, large cache ways are particularly harmful when few functions are placed since large padding cannot be avoided in-between those functions. For instance, for an 8KB way size the binary size is in the range of 2.5 to 3 times the original one if only 10 functions are placed. However, as the number of functions increases such overhead decreases. Finally, note that this sensitivity analysis for functions (Figure 8.3) is also valid for globals as they are analogous problems. Thus, the larger the number of globals or the larger their size, the lower the relative overhead due to SSR.

[1]The number of atoms in the Universe is estimated to be 10^{80}.

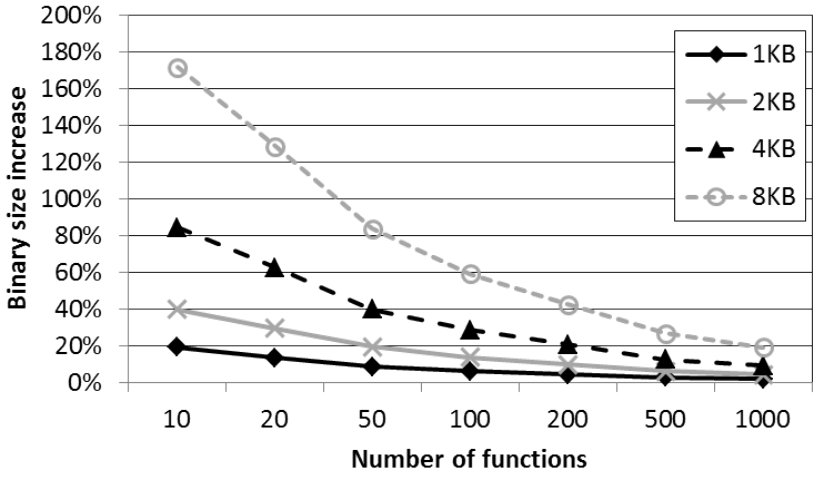


Figure 8.3: Binary size overheads for the sensitivity study varying function number between 10 and 1,000 and cache way size between 1KB and 8KB.

SSR-stack

Regarding *SSR-stack*, it is not stored in the binary. Instead, when functions are called some space may be wasted in the stack due to stack frame padding. As for functions and globals, the largest padding must be smaller than the cache way size. Thus, the overhead is incurred dynamically when functions are called and it can be upper-bounded by the maximum function call depth[1] in the program multiplied by the cache way size. Since programs implementing safety-critical functions do not use complex call structures [Wartel *et al.* (2013)], call depth is typically low (largely below 10 nested calls), thus wasting little memory space. For instance, for a maximum call depth of 5 functions and a cache way size of 2KB, up to 10KB of memory would be wasted dynamically (5KB on average) for *SSR-stack*.

If such overhead is regarded as excessive due to RAM memory availability constraints, the random padding could be added just once at the beginning of the program. The effect would be analogous to packing globals: determinism introduced (all stack frames consecutively placed) remains identical at analysis and deployment, but the degree of randomisation is reduced, thus potentially increasing pWCET estimates.

Note that by randomising at least the starting address of the stack, any further call, including interrupts, will get its stack in a random location. It will not be random w.r.t. other stack frames, but this does not challenge MBPTA as the states observed at analysis time match those at deployment.

[1]The maximum stack depth refers only to the user code stack, excluding system stack which is not modified by our method.

8.4.2 Performance

In the previous chapter DSR has been shown to have moderate or low execution time overheads. SSR introduces fewer instructions (1 per stack frame) than DSR (several for each function/stack indirection) and does not require any initialisation process, as opposed to DSR. Hence, SSR reduces DSR overheads. In particular, SSR execution time overheads are largely below 1% w.r.t. non-randomisation even for programs with a large number of function calls to small functions – the worst scenario for SSR due to stack frame padding.

8.5 Related Work

Software randomisation has been used in the context of security [Li *et al.* (2006)] and bug tolerance [Berger & Zorn (2006)], in addition to the context of WCET estimation we proposed in Chapter 7. In all cases such software randomisation has been performed dynamically.

While timing guarantees obtained with MBPTA have been proven to fit functional safety standards [Stephenson *et al.* (2013)], DSR has some side effects on the functional verification of software since extra pointers and indirections take values unknown at analysis time. To the best of our knowledge, our proposal on static software randomisation is the first solution reconciling MBPTA on COTS hardware and affordable functional verification.

8.6 Summary

MBPTA delivers trustworthy and tight WCET estimates, but poses some requirements on the underlying platform that can be achieved on top of COTS hardware if software randomisation is delivered. Unfortunately, software randomisation challenges functional verification of software against safety standards due to the use of pointers and indirections.

In this chapter we propose a simple solution to override those limitations of *dynamic* software randomisation by providing *static* software randomisation. We show how static software randomisation can be implemented, identifying its advantages (affordable functional verification, simple implementation) and its limitations (some extra storage space required).

Chapter 9

Static Software Randomisation at Source Code Level

9.1 Introduction

Both software randomisation solutions presented in Chapters 7 and 8 require changing system toolchains, i.e. introducing modifications in the compiler and runtime libraries, which mandate their requalification against safety standards like ISO26262 for automotive [AUTOSAR (2006)]. While those solutions have been shown to be successful in enabling MBPTA on top of COTS hardware, and DSR in particular has achieved a high Technology Readiness Level, the cost of toolchain requalification can potentially delay their adoption in some CRTES domains. The reason is that the verification process, including tools, is a costly procedure since 70% of design time is estimated to be spent in verification and this continues to grow as design-size increases [Bailey (2007)].

Moreover, most compilers used in safety-critical industry are proprietary – due to tool qualification requirements –, so the development of the compiler and linker modifications relies completely on the companies selling them. This creates a major obstacle in the wide adoption of MBPTA, since certain platforms, e.g. in automotive, completely lack support for open-source compilers. Finally, the introduced modifications in both the compiler and linker are platform dependent, so they need to be reimplemented for each ISA.

In order to facilitate the wide adoption of software randomisation to enable MBPTA on (virtually) any platform used in CRTES, in this chapter we propose a novel certification-friendly and *Toolchain-Agnostic Software randomisation Approach* (TASA), that randomises the location in which memory objects are defined within the source-code of the program. TASA is based on the principle that there is a direct relation between the order of memory objects in the source-code and

```
int a(int i)
{
 int m=5;
 return i+m;
}

void b(int j)
{
 char k[2]={1,0};
 double f= a(j)+k[0];
}
```

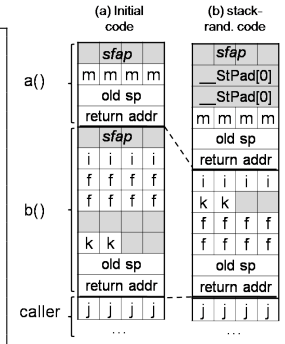
(a) Initial Code Fragment

```
int a(int i);
void b(int j);

void b(int j){
 char k[2]={1,0};
 double f= a(j)+k[0];
 return;
 asm volatile("nop");
 asm volatile("nop");
 return;
}
int a(int i){
 int m=5;
 return i+m;
 asm volatile("nop");
 return 0;
}
```

(b) Same Fragment Code-randomised

```
int a(int i)
{
  volatile double
      __StPad[1];
  int m=5;
  return i+m;
}

void b(int j)
{
  double f;
  char k[2]={1,0};
  f= a(j)+k[0];
}
```

(c) Stack Randomised Code



(d) Stack frame layout for (a) and (c). Alignment padding shown in gray.

Figure 9.1: Code fragments showing code and stack randomisation scenarios.

their memory position, because compilers typically generate the elements of the executable (code, data, etc.) in the same order as they are encountered in the source file. Therefore, by adding functionally-neutral padding code and data and reorganising the declaration of variables and functions across runs, TASA reaches the randomisation properties as previous approaches with the following advantages: (1) TASA provides finer granularity software randomisation, by randomising the internal of the stack frames and fields of compound structures. (2) TASA does not require system toolchain to be modified, as randomisation is applied at source code level. As a result, functional verification and debugging of the system can be performed without the need to modify nor re-certify existing tools. While the tool implementing TASA still needs to be certified, its complexity is lower than the one of a compiler's backend, due to the higher-level (source-code) in which it is applied. (3) Finally, the application of changes at source code level makes TASA to be transparently portable to any platform.

We evaluate TASA on a LEON3 processor used in the space domain. Moreover, we apply TASA on top of two compilers to demonstrate its transparency to different system toolchains. Our results with the industry-standard real-time benchmarks EEMBC Automotive demonstrate that TASA maintains compliance with MBPTA – passing all required statistical tests –, with small impact on memory footprint (7% increase), preserves average performance and provides low pWCET estimates, outperforming the state-of-the-art software randomisation techniques.

9.2 TASA

Software randomisation adds a random padding among memory objects and reorders them from run to run so that a randomised MBPTA-analysable execution time behaviour is achieved. *The challenge lies on doing so while keeping certification friendliness, requiring no change in the application-domain tool chain, keeping MBPTA compliance assessed by passing. i.i.d tests, and keeping the competitive edge in terms of tight pWCET estimates and reduced memory overheads.*

TASA applies *source-code level static software randomisation* (SL-SSR) to enable MBPTA on conventional caches while meeting the constraints presented above. Similar to LL-SSR, the integration of TASA and MBPTA requires TASA generating several images – each with a random allocation of memory objects – that are run on the target platform. Execution times are collected and used to feed MBPTA that delivers a pWCET estimate. At deployment, one of those randomly generated images is used as discussed in the previous Chapter. TASA requires some compiler optimisations to be disabled, incurring minimum impact on average performance as discussed in Section 9.2.7 and experimentally evaluated in Section 9.3.

9.2.1 Executable Structure

In this section, prior to detailing how TASA works, we explain the structure of an executable and how it is related to the source code. Although TASA's principles apply to any binary format, we consider the *elf binary* format [Tool Interface Standard(TIS) (1995)] for the sake of the discussion, which is the most used one. An elf executable comprises four sections: `.text`, `.rodata`, `.data` and `.bss`. Before the program is executed, the linker loads each section in memory, at its corresponding address specified in the executable. The `.text` section contains the executable code of the program. The `.rodata` section contains all read-only program data, which includes global and local static (i.e. local variables that retain their values across function calls) variables declared as `const`, and string literals defined anywhere in the program. Both segments are placed consecutively so that in systems which implement memory protection mechanisms, the entire range of these addresses is write-protected. The `.data` section has the initialised global and local static variables. Finally, the `.bss` section contains global and local static variables which are not initialised or their initial value is zero.

Besides these sections, the binary loader creates at runtime two additional special segments: `heap` and `stack`. The former provides space for dynamically allocated objects and it starts after the end of `.bss` section. The latter provides space for each function's stack frame, where its local variables and arguments live, and it starts at the highest address of the memory space and grows towards the

`heap` (lower addresses). The `heap` segment is ignored in this study since in CRTES dynamic allocation is not allowed.

The elements that compose the executable's sections cannot be arbitrarily placed in memory but certain architectures require the address of memory accesses to be always aligned to the size of the access (e.g. MIPS, SPARC), while others exhibit significant performance penalty when this alignment is not followed (e.g. x86, PowerPC). Hence, compilers generate code that complies with the alignment requirements of the target platform (unless instructed otherwise). For instance, the starting address of each stack frame is aligned to the maximum access size, called *stack frame alignment* (usually 8 bytes), and its size is rounded up to be multiple of this size. This stack frame size adjustment can create empty space called compiler-introduced *stack frame alignment padding (sfap)*. Likewise, each variable is allocated in a memory position multiple of the alignment size as well, e.g. a double precision variable (8 bytes) must be allocated in a memory address multiple of 8, while a `char` variable has no placement requirements since every address location is multiple of 1 byte. This empty space between consecutive variables is called *alignment padding (ap)*. This padding plays a fundamental role in our proposal.

9.2.2 Code Placement Randomisation

TASA performs code randomisation via two mechanisms.

Function ordering. Code randomisation relies on the appreciation that, by default, the location of functions in the binary takes place in the same order that they are encountered in the source file, unless certain compiler optimisations are enabled [McFarling (1989)]. As a result, randomising the order of functions in the source file achieves the goal of randomising the placement of the functions in the object file. This is illustrated in Figure 9.1. In the original file (Figure 9.1a), function `a()` precedes function `b()`, so the compiler generates code in the same order in the object file. Figure 9.1b, shows the corresponding randomised code, in which the functions are swapped. Moreover, in order to guarantee correct cross function dependencies, function prototypes are introduced in the beginning of the file (if not present).

Function Padding. Although random function reordering provides different mappings in the instruction cache, the number of different cache layouts is limited by the number of function permutations.This is mitigated by artificially increasing the size of each function by a random *padding*. TASA adds padding in the form of an arbitrary number of additional instructions at end of the function as shown in Figure 9.1b. The number of added instructions ranges from 0 up to the number of instructions that fit in a cache way since this covers all the potential mappings of instructions to cache sets. The new inserted instructions have no functional or

timing effect on the executed code, which is achieved by using *nop* instructions. C programs can directly call assembly instructions using the `asm` directive. While in general using assembly instructions limits source code portability, note that this particular instruction does not harm the portability of the technique due to its ubiquitous nature. Moreover, the `volatile` qualifier in the asm instruction is required to prevent the compiler from removing the introduced assembly instructions during its optimisation passes. Further, in order to avoid that the introduced instructions affect functions' execution time, we ensure that the new inserted code is never reached by introducing a `return` statement (if not present in the code) with a return value compatible with function's return type (e.g. function `b()`). In the case that the function features more than a single return point, this padding is applied only to the last one, since the purpose to pad the size of the function, without any functional effect.This solution requires dead code elimination and unreachable code elimination optimisations of the compiler to be disabled, otherwise the added code would be automatically removed by the compiler.

9.2.3 Stack Frame Randomisation

We achieve stack randomisation through 2 different complementary and combinable methods.

Stack Padding. Since the size of the stack frame is determined by the number and the size of the function's local variables and the arguments of the functions that it calls, we can randomise the stack by introducing a randomly sized local array in the list of local variables, see function `a()` in Figure 9.1c. We use the *volatile* qualifier to instruct the compiler not to perform any optimisation on these variables. In order to effectively increase the stack frame, even when the frame size is rounded up to the alignment size, the array is declared as `double`. Similarly to code placement randomisation, the size of the array, and therefore the padding introduced, is randomly sized up to the size of a cache way.

Figure 9.1d shows how the stack frame of `a()` changes after applying stack randomisation. In the original code, the stack frame contained only a 32-bit variable plus the always included return address and the old stack pointer. Assuming a 32-bit environment, the size each of the return address and the stack pointer is 4 bytes, therefore the compiler rounds up the stack frame size to 16 bytes, in order to comply with the stack frame alignment requirements, instead of using a 12 byte stack frame. In the randomised scenario, the stack frame also holds a double variable increasing the stack frame size, which with stack alignment padding reaches 24 bytes size. Note that if instead of double, the type of `__StPad` were float or integer, the desired effect would not be achieved, and in both cases the stack frame of `a()` would have the same size.

Local Variable Declaration Order. The stack can also be randomised by chang-

```
const char msg[]="OK";
long l;
long m=3L;
unsigned int f;
const double pi=3.14;

int c(void){
  static int i=1;
  if(i==MAX){
    i=0;
    printf("reset");
  }
  return i++;
}

void d(int i){
  static long time;
  static long ticks=1L;
  time = clock();
  ticks++;
  printf("%l ms", time);
}
```

(a) Initial Code Fragment

```
const double pi=3.14;
const char msg[]="OK";
long m=3L;
unsigned int f;
long l;

void d(int i){
  static long time;
  static long ticks=1L;
  time = clock();
  ticks++;
  printf("%l ms", time);
}

int c(void){
  static int i=1;
  if(i==MAX){
    i=0;
    printf("reset");
  }
  return i++;
}
```

(b) Global & Static Variable rand.

Figure 9.2: Code fragments under various data randomisation scenarios.

ing the declaration order of local variables, as they are allocated on the stack based on their position from the stack pointer. Such a fine-grain stack randomisation, which cannot be achieved by the existing software randomisation solutions, is of particular interest because it achieves randomisation of *intra-object* [Kosmidis *et al.* (2013c)] conflicts (*intra-stack* in particular). Figure 9.1d shows the impact that the order in which local variables are declared in function `b()` defined in Figure 9.1c has on the stack frame. The compiler introduces an alignment padding (sfap) between `k` and `f` to ensure that `f` is placed on an address multiple of 8. Such alignment padding however may not be required when shuffling the local variable declaration as shown in Figure 9.1d.

9.2.4 Program Data Randomisation

Global data which includes variables defined in the global scope of a program, static variables defined in each function and string literals, reside in the corresponding sections `.rodata`, `.data` and `.bss`, depending on whether they are constants and have/lack initial values.

Figure 9.3a shows the content of sections `.text`, `.rodata`, `.data` and `.bss` for the initial code fragment shown in Figure 9.2a, including the corresponding alignment. Despite compiler's optimisations, the location of each variable in the

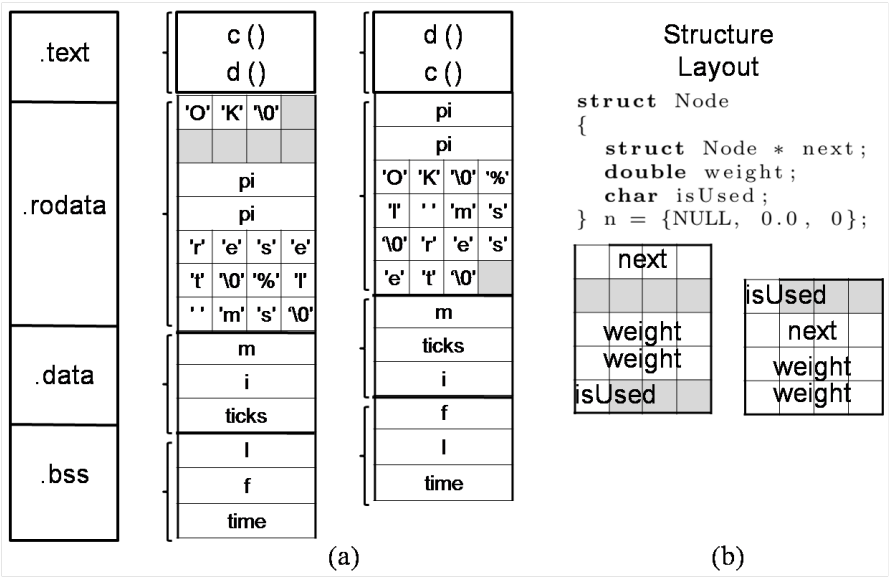


Figure 9.3: (a) Memory layout for corresponding source code fragments from Figure 9.2. (b) Struct memory layout.

binary is heavily dependent on its *relative position in the source code file with respect to the variables mapped in the same binary section.* In other words, swapping the order of two variables will only affect the mapping if it changes their relative position regarding the rest of the variables mapped in the same section. For example swapping variables `l` and `m` in the source code shown in Figure 9.2a would not change the mapping in the binary which is displayed in Figure 9.3a as they reside in different data sections, `.bss` and `.data` respectively. On the other hand swapping `msg` and `pi` would, as both variables reside in the same section (`.rodata`).

TASA guarantees that variables from different sections are shuffled by grouping variables belonging to the same section first before shuffling each group individually. Similarly to the stack frame, changes in the order of symbols in the sections may produce changes in the alignment padding. This affects the position (and size) of the subsequent variables in the same section and the position of the symbols in the following sections. Figure 9.2b shows the source code after applying program data randomisation, while its corresponding memory layout is shown on the right part of Figure 9.3a.

Static variables and string literals: The position of these types of variables in the corresponding section is bound to the position in the source code of the function they are declared in, e.g. as shown in the string literals used in printf in Figures 9.2 and 9.3. If `c()` and `d()` were not swapped, their position in `.rodata`

would remain unchanged. The same effect happens for static variables `i` and `ticks`. Hence, static variables and strings cannot be randomised freely, unless code randomisation is enabled too. This is an important difference with LL-SSR we proposed in Chapter 8, where these variables can be shuffled independently from the code of their function. However, in large applications which contain many functions and static data, the memory layout is not meaningfully affected by this binding.

9.2.5 Compound Structure Randomisation

None of the existing software randomisation solutions manages efficiently compound constructs, like programming language structures (Figure 9.3b). This is because the access structures' member variables are allocated in a relative position (offset) from the starting address of the structure, based on their declaration order in the structure definition and taking into account each member's type alignment requirements. This offset is hard-coded into the binary, and so neither DSR nor LL-SSR can modified structures' member positions Instead, TASA can shuffle structures' members in order to randomise data access patterns when used in arrays, and intra-object conflicts, provided the size of the structure is bigger than a cache line. Similarly to stack randomisation (see Section 9.2.3), due to members' alignment padding, the size of each structure depends on the particular ordering of its members, which in turn affects the memory layout. This is illustrated in the example of Figure 9.3b. This process requires the aggregate initialisator (if any) to be shuffled respectively and the `__packed` attribute, which prevents alignment padding, to be absent from the the structure's definition.

9.2.6 Multi-source Binaries

When a program with multiple sources is compiled, each file is first compiled to produce an object file. Each object file is comprised by the same sections as the final executable. During the linking phase, these sections are joined together, in the order that the object files are passed to the linker, to form the sections of the final executable. Hence, if TASA were applied in each source file independently, the elements of each section could only be randomised with respect to the elements in the source file in which they are declared. To avoid this problem, TASA merges all files in a single source file, taking into account *symbol linkage*. Then the previously described randomisations are applied with global scope.

9.2.7 Compiler Optimisations

In order TASA to be applied, a few compiler optimisations need to be disabled as identified in the previous sections. Concretely, function/data reordering and dead-code/unreachable code elimination.[1] These optimisations can be divided in two groups: a) performance optimisations and b) size optimisations. All but one optimisation, fall in the latter group. In particular, the only performance enhancing optimisation which is disabled is function reordering for performance. While this can be effective for small code, its effectiveness is reduced when the number and the size of functions is increased. This is the case in modern safety critical systems, since the total software size for avionics systems exceeds 100 MBs [Edelin (2009)]. Moreover, this process is NP-complete for a large number of functions [Mezzetti & Vardanega (2010)], therefore can lead to extremely long compilation times. For this reason, this optimisation is typically disabled in such systems.

The rest of the optimisations have no performance cost, but memory size cost. These optimisations try to find function and data ordering that minimise the overall memory footprint. Similarly, these optimisations don't scale well for large code bases, increasing significantly the compilation times.

Disabling dead-code/data and unreachable code elimination, if they are effective, impacts size only. Safety critical systems must execute only code that is verified at deployment. Unremoved dead-code/data as well as the padding introduced by TASA are non-functional elements that are never executed or used. Thus, this constraint is still met.

Section 4.3 shows that disabling those optimisations on EEMBCs has no performance or memory impact.

9.3 Evaluation

In this section we assess TASA in terms of average performance, tightness of pWCET estimates, memory overhead, compliance with certification and transparency to the underlying toolchain.

9.3.1 Experimental Setup

We developed a source-to-source compiler for ANSI C programs implementing all software randomisation techniques described in Section 9.2. In order to demonstrate the ability of TASA to work independently of the particular industrial toolchain, we use two different compilers to compile the TASA-transformed code: gcc-4.4.2 and vanilla llvm 3.1.

[1] The use of `volatile` in code/stack randomisation is not included in this list, since it prevents only the removal of the TASA introduced padding.

Several compiler flags are used to maintain transformations implemented by TASA as shown in Section 9.2.7: (1) `-fno-toplevel-reorder` instructs the compiler to generate functions and globals in the order encountered in the binary; (2) `-fno-section-anchors` prevents placing static variables used in the same function, in nearby locations in order to be accessed with a single base and (3) `-fno-dce` disables dead-code elimination.

Benchmarks. We use the EEMBC Autobench benchmark suite [Poovey (2007)] as explained in Section 3.3.

Platform. We focus on a FPGA version of the Sparc v8 LEON3 [Cobham Gaisler (2005)] processor developed by Cobham Gaisler, a widely-used processor in the CRTES domain, especially in aerospace. It features a 4-way set associative instruction cache with 128 sets per way and 32-byte cache lines, and a 4-way set associative data cache with 256 sets and 16 bytes per cache line. In order to limit the memory overheads of TASA we select the maximum random padding to be equal to 1/20th of each cache way size, that is at most 50 instructions and stack padding 50 doubles. The FPGA implements a quad-core LEON3 processor, although we execute applications in a single-core mode due to the single-threaded nature of the applications.

Methodology. We follow the same methodology introduced in Chapter 3, with the only difference that we use a real platform instead of the simulator. For each benchmark we perform 1,000 passes of the original source code with TASA, in order to generate an equal number of binaries with different layouts. Each binary is executed once and its end-to-end execution time is collected with *grmon* [Cobham Gaisler (2005)] accessing the cycle performance counter of LEON3.

Reference Software Randomisation Technique. We compare TASA with the dynamic software randomisation technique we proposed in Chapter 7. The compiler pass of the original open source tool works only on LLVM 3.1, for this reason we used that old version of LLVM. For each benchmark we perform all software randomisation techniques described in Chapter 7 and collect 1,000 end-to-end execution times. In addition to porting Stabilizer on Sparc V8 (supported only x86, x86-64 and PowerPC), we modified it to make it more friendly to the CRTES domain and directly comparable to TASA, with the modifications improving Stabilizer results. In particular, we implemented: a) eager function relocation – relocation of all functions at start up, not at the moment of their invocation –, b) non-trap invoked relocations, c) no re-randomisation and d) reduced padding for code and stack in values smaller than pages. We show that even with these modifications, the overheads of DSR are higher than those of TASA.

Since TASA is functionally equivalent to the static randomisation proposed in the previous Chapter (LL-SSR), we do not provide a comparison with it. In fact TASA provides identical benefits, but also exhibits the following advantages over LL-SSR: No changes are required in the original toolchain, while LL-SSR requires modifications in both the compiler and the linker. Moreover the level of randomisation which can be achieved is finer as a) stack randomisation can be enhanced with intra-stack frame randomisation, instead of the monolithic stack frame displacement provided by DSR, introduced in Chapter 7and b) TASA is able to provide intra-structure randomisation which has effects in every data aspect (global/stack). On the downside, TASA provides coarser grain randomisation of static variables (which are less frequently used than stack frames and structures) and string literals.

9.3.2 Certification Compliance and Transparency

In CRTES applications follow a number of software design principles to meet safety standards. For example in the automotive domain the standard ISO26262 [International Organization for Standardization (2009)] requires among others: a limited use of pointers, no use of dynamic objects, and no hidden data or control flow. This is particularly true for the highest safety levels (e.g.ASIL-C and ASIL-D). Similar standards exist for avionics [RTCA and EUROCAE (1992)], aerospace [JPL (2009)] and other critical systems [MISRA (2013)]. From the description in Section 9.2 it follows that TASA is compliant with all these principles. From Section 9.2 it also follows that TASA is transparent to the underlying toolchain, which we show by showing results of TASA on top of gcc-4.4.2 and vanilla llvm 3.1.

9.3.3 Impact of Optimisation Disabling

In order to evaluate the impact of the compiler optimisations that we need to disable for TASA to be applied, as summarised in Section 9.2.7, we compared the original non-randomised application with optimisations enabled and disabled in terms of both memory footprint and performance. We have observed that for gcc there was no effect of those optimisation neither in memory consumption nor in performance. For llvm the result was exactly the same; none of the 3 dead code elimination (dce) passes (dce, advanced dce and global dce) removed any dead or unreachable code[1]. This can be explained by the fact that EEMBC have a small/controlled codebase which does not contain any dead code.

[1]To our knowledge LLVM doesn't have any reordering pass for code/data.

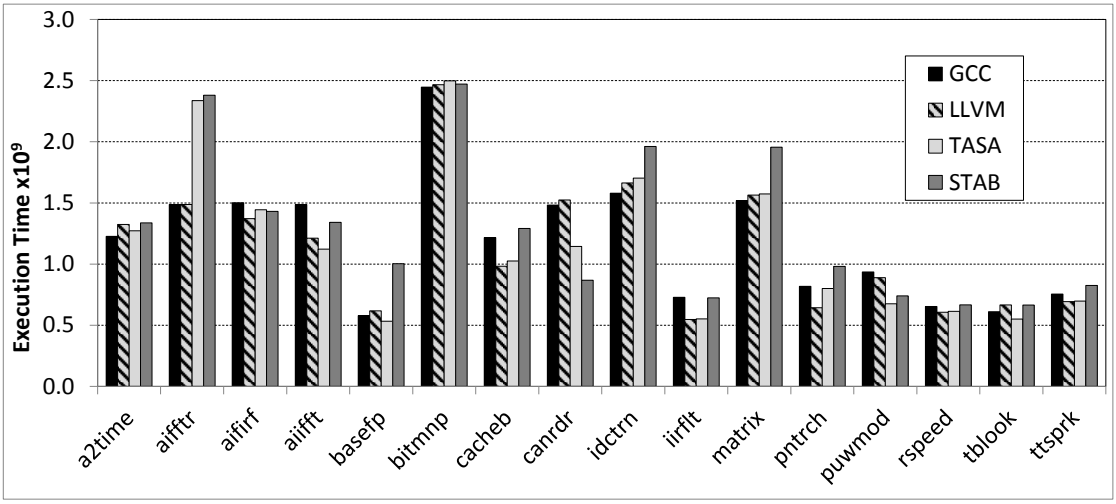


Figure 9.4: Average execution time measured in processor cycles for TASA and DSR (STAB).

9.3.4 Average Execution Time

Besides WCET, average performance is important in CRTES to optimise non-functional metrics such as power and energy. Figure 9.4 shows the average execution time for EEMBC benchmarks when compiled with GCC, LLVM, TASA (using the LLVM compiler) and Stabilizer (using the LLVM compiler). For both variants, all randomisations that each tool supports are enabled. Note that for TASA we provide execution times only with LLVM for a straightforward comparison with Stabilizer, which is based on LLVM. In general both TASA and Stabilizer provide average execution times close to the default generated code by GCC and LLVM, with TASA being on average within 0.4% of LLVM's average performance. In two cases, the average performance of both randomised configurations is significantly different to that of the non-randomised one such as `aifftr` (worse) and `canrdr` (better). This is related to the default memory layout selected by the compiler, which happens to be very good or very bad compared to the set of potential memory layouts, which randomisation can explore. Finally in almost all cases, the execution times of Stabilizer are longer than for TASA, with significant differences at times (`basefp`, `matrix`). The reason is due to Stabilizer's execution time overhead: At start-up the runtime performs some required operations before the execution of the program such as the code relocations, while other additional stack-randomisation are performed at runtime. Additionally, the increase of the `.bss` section due to the introduced metadata (see Section 9.3.5) induces a significant performance degradation issue, since this section needs to be zeroed-out before the execution of the system. While this operation is optimised in desktop

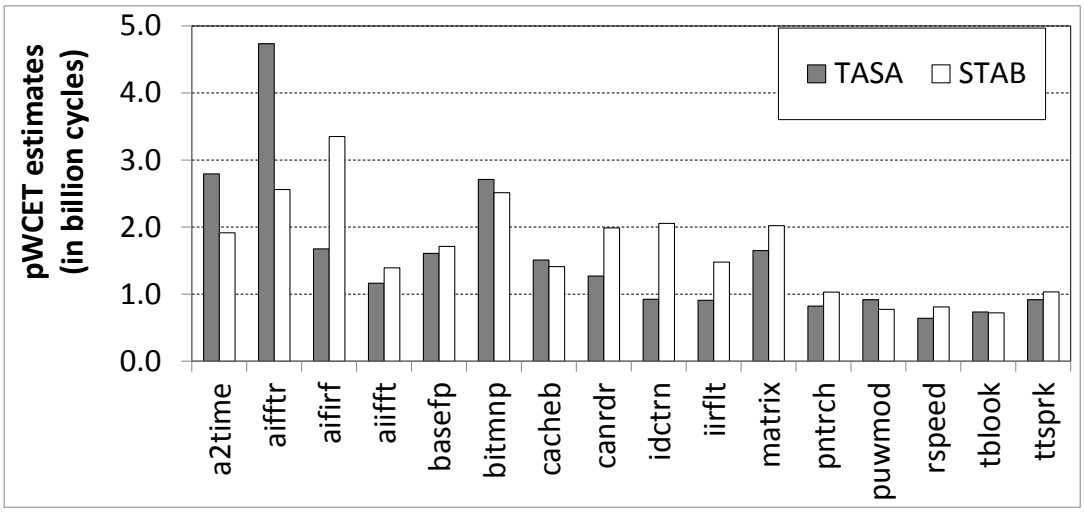


Figure 9.5: Worst Case Execution Time for TASA and Stabilizer

systems by techniques like copy-on-write, in an embedded system without operating system like our target, it cannot. Therefore the bigger the `.bss` is, the longer this operation takes.

9.3.5 pWCET Estimates and MBPTA Compliance

Since TASA targets real-time systems, it is vital it results in tight and trustworthy pWCET estimates. pWCET estimates are obtained as explained in Chapter 3. Instead of showing the pWCET curve as presented in Figure 1.2, we show the pWCET estimates obtained with the different techniques for a cut-off probability of 10^{-15} per run. This value has been chosen since it has been shown appropriate for applications with the highest criticality levels in several domains such as avionics [Wartel *et al.* (2013)][Wartel *et al.* (2015)].

As explained in the Background, the safe application of MBPTA requires the obtained execution time observations to be modellable with independent and identically distributed (i.i.d.) variables. This can be assessed by using the corresponding statistical tests discussed in Section 3.2.

In Table 9.1 we show the results of those tests for TASA. We observe that in all cases the results of the tests confirm that the execution time observations take from the different binaries, each with a randomised layout, can be modelled with i.i.d variables, making TASA compliant to MBPTA.

Figure 9.5 shows the pWCET for EEMBC benchmarks, at a cut-off probability of 10^{-15}, with all possible randomisations enabled w.r.t. the actual execution time on LLVM with no randomisation. We observe that in the vast majority

Table 9.1: I.I.D. tests for TASA

benchmark	Identical Distribution	Independence
a2time	0.67	0.53
aifftr	0.08	0.16
aifirf	0.33	1.33
aiifft	0.93	0.79
basefp	0.99	0.88
bitmnp	0.83	0.23
cacheb	0.80	1.32
canrdr	0.44	0.88
idctrn	0.65	1.56
iirflt	0.83	0.24
matrix	0.25	0.50
pntrch	0.99	1.13
puwmod	0.99	1.13
rspeed	0.21	0.50
tblook	0.99	1.01
ttsprk	0.66	0.96

of the benchmarks TASA provides lower (6% on average) execution times than Stabilizer, following the same trend as for average performance, due to the start-up and runtime overhead of dynamic software randomisation. However, in few cases (`a2time`, `aifftr`) the pWCET of TASA is higher than Stabilizer's one despite the lower average performance. This occurs because TASA can explore some rare memory layouts (3% of the explored ones) with much higher execution time than the average one, that Stabilizer was not able to generate, due to its coarser-grain randomisation.

9.3.6 Memory Overheads

Real-time systems are resource-constrained, especially in terms of memory, so low-memory requirements are essential. Figure 9.6 shows the increase in memory consumption of the `text` (Figure 9.6(a)), `data`((Figure 9.6(b)) and `bss` (Figure 9.6(c)) segments for the EEMBC Automotive benchmarks when using TASA with gcc and llvm (labelled as *TASA-GCC* and *TASA-LLVM* respectively), and Stabilizer with LLVM (labelled as *LLVM-STAB*). Each software-randomised setup is normalised to the same non-randomised configuration, e.g. TEXT TASA-GCC provides the relative increase in the text segment when TASA is applied over the non-randomised one, when in both cases gcc was used for compilation.

Code Segment. Code padding increases the code footprint of the application. As shown in Figure 9.6a, TASA introduces an overhead less than 1.13% on average

when used with gcc (*TEXT TASA-GCC*), applying a random padding up to 50 nop instructions per function (200 bytes). Using TASA with LLVM (*TEXT TASA-LLVM*) this overhead is only 7%, because the generated code with gcc is more compact, therefore the relative increase is smaller.

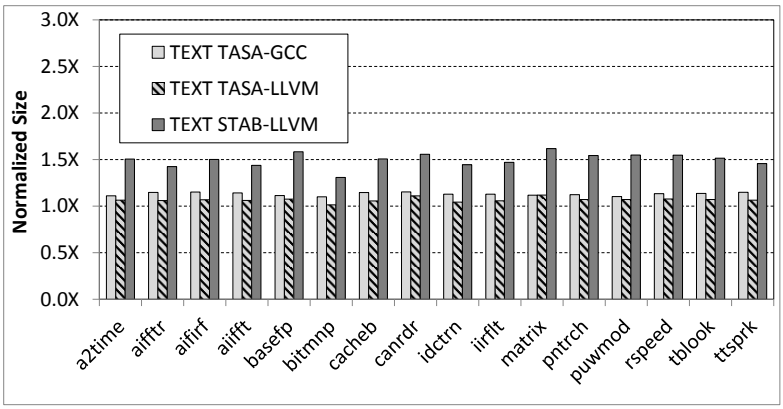
Stabilizer however introduces a significant memory overhead. In particular Stabilizer doubles the size of the executable on average (*STAB*). This increase comes by the runtime code which is linked with the application. Since the runtime is written in C++ and the code is heavily-based on templates, this contributes significantly in the increase. Note that the original code size of the applications is in the range of 70-100 KBs, therefore linking with a library of 130 KB adds an important overhead. Moreover, Stabilizer increases the memory consumption of `text` segment at run-time, during which the code relocation takes place once per function. In original Stabilizer, each function can have a padding up to one page, therefore two pages (8K) are allocated. The runtime places the function in a random offset inside a page, so that random placements are not biased towards starting offsets near the beginning of the page. This implementation results in an increase of the memory consumption of code of 6.5× for these benchmarks. Obviously this implementation is inappropriate for CRTES which are memory constrained and increases linearly with the number of functions of the program.

In order to perform a fair comparison, in Figure 9.6a we show the code memory consumption of the dynamic software randomisation using a random padding equal to TASA (up to 200 bytes).

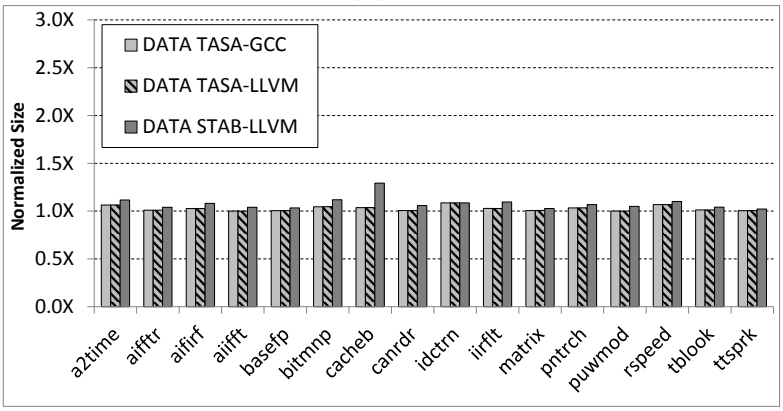
It is important to note that the original implementation of Stabilizer, as described in Chapter 7 has a lower average memory consumption for code, because it relocates only the functions that are actually called at runtime. While this is true for general purpose programs, this creates issues in CRTES, which require worst-case guarantees for the memory consumption. Allocating all functions at the initialisation phase of the program satisfies this requirement. Besides, in critical real-time systems there is a requirement not to use source code of functions which are not verified (and so never called) in the final binary. Therefore for real-time systems, both Stabilizer implementations (vanilla and our modified for the comparison) have the same (worst and average) code memory consumption, which is 75% bigger than TASA.

Data Segments. Per-segment results show the following.

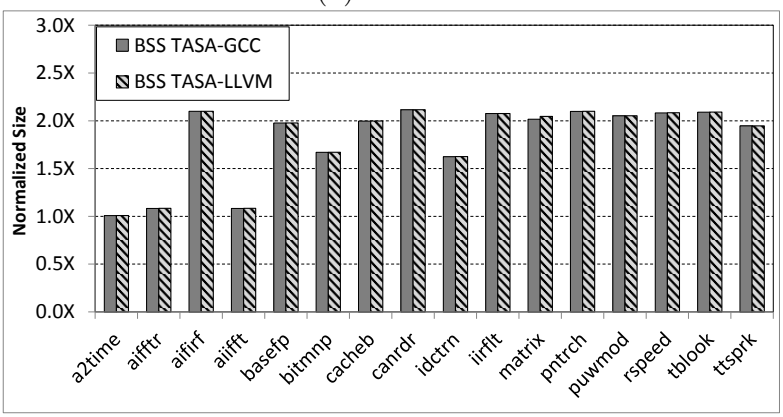
Stack. For the stack size, results not shown in Figure 9.6, TASA increases the stack space used per function around 25% on average, using a stack padding size up to 50 doubles (400 bytes). When local variables randomisation is enabled there is a variability of the stack size of ±1%. Original Stabilizer adds a padding between stack frames up to a page, similarly to code. Again in order to achieve the same effect with TASA and for the purpose of a fair comparison, we modify this



(a) code



(b) data



(c) bss

Figure 9.6: Memory overhead for different binary sections. Results are normalised to the corresponding toolchain (gcc or llvm) without software randomisation. Values for TASA are the average for all binaries.

size to 400 bytes. While the maximum stack consumption in both Stabilizer and TASA is the same, Stabilizer introduces 8 instructions in the function's prologue and epilogue to adjust the stack pointer, leading to a 35% increase in code on average.

Data. For the `data` section, the overhead of TASA for most benchmarks is less than 7% for both toolchains (Figure 9.6b), while Stabilizer's is 8%.

Bss. For the `.bss` (Figure 9.6c) TASA's overhead is 80% on average, but its contribution in the total footprint is small, since in represents less than 10% of the memory footprint in our benchmarks. The reason for the increase is that EEMBCs contain large uninitialised arrays which are used to simulate data output. Note that in a real system those arrays would not exist. Randomising the location of these large arrays results in big size differences due to the alignment padding, as we have explained. Stabilizer's impact on `.bss` is enormous and it is not depicted in the Figure. In absolute numbers the bss increases from 2KB to 33MB. This increase comes from the functions' metadata which are generated by Stabilizer's compiler pass and used by the runtime for the relocations as well as the data structures that the runtime uses.

Overall, taking into account both code and data segments, TASA incurs a 7% increase on the memory footprint of EEMBC Automotive benchmarks.

9.4 External Results

Unlike the previous contributions of this thesis that have been first evaluated on a simulator and later on real hardware, TASA has been designed from the very beginning to work on real hardware. The reason is that at the moment of TASA's inception, software randomisation solutions as described in the previous chapters had reached a mature state, especially the dynamic software randomisation variant. In the PROXIMA project we considered 3 COTS hardware platforms: LEON3, P4080 and AURIX, in order to cover processors that are well established or considered for future use in the CRTES domains represented by the project's industrial partners in the domains of aerospace, avionics, railway and automotive. We enabled MBPTA on both LEON3 and P4080 using dynamic software randomisation, by porting and adapting our proposed solution in the open source LLVM infrastructure, which supports both SparcV8 and PowerPC ISA respectively.

However, this was not the case for AURIX Tricore. AURIX, a microcontroller used extensively in the automotive market, supports only three proprietary compilers: Tasking from Altium, and HighTec and Diab from WindRiver. All compilers are only distributed in binary form and require an expensive license. Tasking and Diab compilers are completely closed source, while HighTec is gcc-based. Although the source code for an old version (last updated in 2008) of the compiler is available from [HighTec (2008)] as required by the GPL licence of gcc, that version is

not compatible with recent boards that are available in the market, like the ones used in the project. Therefore, despite the success of our previous software randomisation solutions with several real boards, they could not be used on AURIX, due to the unavailability of the compiler sources.

Moreover, even in the case that the sources were available, AURIX has limited support for self-modifying code and features a very small (32KB) RAM memory – Local Memory Unit (LMU) in AURIX architecture terminology –, which is compensated by a large flash memory (4MB).

As a consequence, TASA has been designed to specifically address a real need of enabling MBPTA on the AURIX TriCore platform [Infineon (2012)], which was not possible by our previous software randomisation proposals, due to the platform and ecosystem peculiarities.

Therefore, in addition to the TASA evaluation on LEON3 presented in the previous section, TASA has been also evaluated on AURIX using the free evaluation version of the HighTec compiler, the open source real-time operating system ERIKA [Evidence (2012)] and an automotive case study developed in the CONCERTO [CONCERTO (2016)] project. The result of this work has been published in [Kosmidis *et al.* (2016a)] and confirms that TASA is both toolchain-agnostic and completely portable across different hardware platforms. Moreover, it shows that MBPTA can be enabled using software randomisation, passing all statistical tests, even on a completely deterministic processors such as AURIX, which exhibits zero jitter when the same binary is executed under the same input set.

In addition, it is demonstrated that TASA provides a uniform distribution of the code in the instruction cache sets, validating that indeed the source code modifications impact the placement in the cache. Finally, the MBPTA curves derived for each function of the case study provide 5-15% tighter pWCET estimates than current industrial practice (e.g. applying an 20% engineering margin over MOET) even for 10^{-12} exceedance probability per run.

9.5 Related Work

Many studies have shown that the program's memory layout affects significantly its performance in modern processors [McFarling (1989)] [Gloy & Smith (1999)] [Mytkowicz *et al.* (2009)]. [Mytkowicz *et al.* (2009)] has shown that the linking order of object files affects significantly the execution time of the program. Moreover, it has demonstrated the impact of small changes of the memory layout in performance, eg. environmental variables. Based on this observation [Curtsinger & Berger (2013)] used software randomisation to isolate this effect from the actual performance benefit of various compiler optimisations.

Software Randomisation has found also various applications in the literature.

[Xu *et al.* (2003)] proposed software randomisation for security, to avoid attacks exploiting the static memory layout of a program (eg buffer overrun). This idea has been implemented in commodity operating systems to enhance security [Bhatkar *et al.* (2003)] [Shacham *et al.* (2004)]. Similarly [Berger & Zorn (2006)] used software randomisation for reliability to avoid crashes due to programming errors.

9.6 Summary

MBPTA's promising results can only be obtained in those platforms adhering to MBPTA's requirements. Prior attempts to make systems compatible with MPBTA require changes in different parts of the toolchain stack, such as the compiler, linker, runtime or even hardware, thus increasing the cost of its adoption in industrial environments due to additional costs related to the development and certification of the modified toolchain. We overcome this limitation by proposing TASA, a software randomisation technique applicable in the source level of the application, and therefore portable to any platform. We show experimentally using EEMBC benchmarks running on a COTS processor from the CRTES domain, that TASA is not only superior than existing software randomisation techniques in terms of memory and execution time overhead for real-time systems, but also provides finer-grain randomisation.

Chapter 10

Path Upper-Bounding for MBPTA

10.1 Introduction

In contrast to the earlier chapters, which presented hardware and software solutions to enable the use of cache memories in the context of MBPTA, in the following two ones we introduce solutions dealing with timing analysis aspects which arise from the use of caches in this timing analysis method.

As discussed briefly in the Introduction and in the Background, MBPTA provides pWCET estimates that upper-bound the execution time of those paths exercised with the input vectors provided by the user[1]. Unfortunately, in general, the user cannot determine all those paths leading to the highest execution times and hence, may not provide input vectors that exercise them. Typically, timing analyses rely on the ability of the user to provide loop bounds and/or generate input vectors exercising the highest loop bounds, which is already a challenging task. However, conditional control-flow constructs (CFC) such as *if-then* and *if-then-else*, are not easy to bound. This is so because detailed knowledge about the processor state (e.g., cache state) is required to determine which path (i.e. sequence of branches) across different CFC impacts execution time the most, with the number of paths growing exponentially with the number of *dynamic* CFC[2].

In this chapter we propose *PUB*, a path upper-bounding method that extends MBPTA such that the provided pWCET estimates upper-bound any path in the program, even when the input vectors do not exercise the worst-case path. To

[1] 'user' refers to the system designer, integrator or whoever uses MBPTA.

[2] Dynamic CFC stands for the sequence of CFC traversed during execution. For instance, if there is one static CFC in the code but it is executed N times within a loop, it leads to N dynamic CFC producing up to 2^N different execution paths.

that end, *PUB generates an extended version of the program* by adding instructions in the different branches of CFC so that the execution time of any path of the extended program upper-bounds the execution time of all paths in the original program. Interestingly, the extended program can be used only to generate the pWCET at analysis time, while *at deployment time the original unmodified program can be used*. We also introduce a variant of *PUB* that reduces the overhead of the original version for some specific CFCs.

Unlike MBPTA that requires full-path coverage to provide the pWCET for a program, *PUB* requires few input vectors, i.e. end-to-end traversals of the program. This requirement is much less than basic block coverage, which in fact is required for functional verification. For instance, modified condition/decision coverage (MC/DC) is required for the highest design assurance level (DAL), DAL A, in avionics. Hence, in the general case, *PUB* makes no input vector be required for timing verification of the system beyond those input vectors provided for functional verification. This significantly reduces the cost of applying the analysis technique.

PUB builds upon the observation that caches deploying random placement and replacement, introduced in Chapter 5, a.k.a time-randomised caches, required for MBPTA are much less history-dependent than conventional caches based on, for instance, modulo placement and least recently used (LRU) replacement. Cache history dependence on PTA-compliant hardware platforms is made probabilistic and the dependence between the memory address assigned to a particular piece of data and its assigned set in cache is broken. As a result, upper-bounding the effect of the different branches of CFC in the state of a time-randomised cache can be done with lower complexity than for a time-deterministic cache.

Our results show that *PUB* provides pWCET estimates that upper-bound the execution time of every path in the program, while increasing pWCET estimates only by 11% for EEMBC benchmarks and 5% for Mälardarlen with respect to the pWCET estimates provided by MBPTA using user-provided input vectors[1]. Code size of the extended version of the program grows on average by 26% and 3% respectively for EEMBC benchmarks and Mälardarlen, and such growth is related to the number of CFC and their degree of nesting.

10.2 Path Coverage

One of the most challenging steps in applying MBPTA and, in fact, any measurement-based technique, is the generation of test data. The user must provide a range of input data to the program, designed to stress the program and produce worst-case

[1]For some benchmarks the actual paths exercised may not include the worst path, which plays against *PUB* since the execution time difference between the exercised paths and the actual worst one is deemed as *PUB* overestimation.

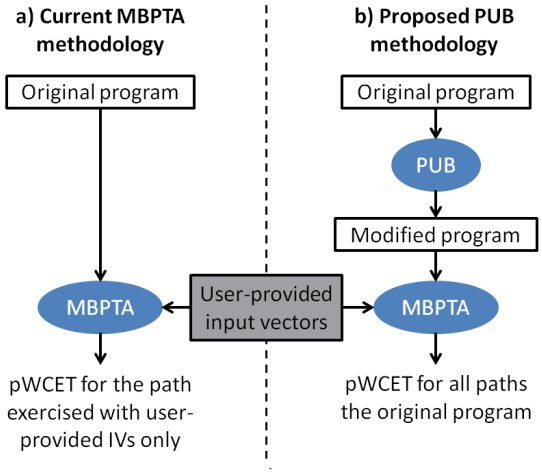


Figure 10.1: Current and proposed methodologies based on MBPTA.

behaviour. The choice of the data may affect the timing behaviour of the software, hence properly selecting test vectors ensures trustworthiness and efficiency of the overall analysis. It is also the case that common code coverage criteria from the domain of functional testing, including Random Testing, Basic Block Coverage, Condition/Decision Coverage and Modified Condition/Decision Coverage (MC/DC), can underestimate the WCET in the context of measurement-based techniques [Bünte *et al.* (2011)]. In particular, MBPTA depends on the input vectors provided by the user in terms of (1) loop bounds and (2) WCET-relevant execution paths. Providing input vectors with WCET-relevant loop bounds can be regarded as attainable for the user in general given that the highest values are the WCET-relevant ones. However, determining which execution paths – out of the combination of all branches of all dynamic CFC – are the ones leading to the WCET is, at best, very difficult. Further, generating input vectors exercising those paths can be regarded as unattainable in the general case.

The analysis of the timing behaviour of caches and its interaction with control-flow analysis have been deeply analysed for deterministic systems [Grund (2012)] [Wilhelm *et al.* (2008)]. Time-randomised caches remove any dependence on the location of objects (e.g. code, libraries, OS, heap, etc.) in memory and so, on the particular cache location in which they are located, which simplifies the analysis of cache addresses across different paths. Despite that, the dependence among different accesses is not completely removed, but that dependence is made probabilistically modellable. In Section 10.3 we present how *PUB* builds upon the features of time-randomised caches to allow upper-bounding different branches of CFC at low cost.

10.3 Principles of *PUB*

PUB allows MBPTA to derive pWCET estimates that probabilistically upper-bound the execution time of any path in the program even when the user-provided input vectors do not exercise the worst-case path. The picture on the left in Figure 10.1 shows the MBPTA flow, where end-to-end execution time measurements are collected for paths exercised with the user-provided input vectors. pWCET estimates obtained are **only** guaranteed to upper-bound the execution times of exercised paths. Nothing can be stated about any other execution path. Instead *PUB*, see right picture in Figure 10.1, operates either on the original source code[1] or the object code adding instructions in the different branches of CFC such that, regardless of which particular branch is traversed in each dynamic CFC in the modified code, the pET for the extended program upper-bounds the pET of the original program for any traversal (i.e. path) of the conditional constructs. That is, if the original program has EP^{org} different execution paths and the extended has EP^{ext}, the following equation holds, where $pET(p)$ is the pET of the execution path p:

$$\forall(ep_i \in EP^{ext}, ep_j \in EP^{org}) \ : \ pET(ep_i) \geq pET(ep_j) \tag{10.1}$$

Let us assume a CFC with two conditional branches, being the sequence of instructions executed in each branch IS_{org}^{left} and IS_{org}^{right} respectively. The main idea behind *PUB* is adding new operations $\mathcal{O}$, both core operations (e.g. add) and memory operations, with neutral impact on functionality and that result in the extended instruction sequences IS_{ext}^{left} and IS_{ext}^{right}. Added operations $\mathcal{O}$ ensure that the pET of the sequence IS_{ext}^{total} consisting of any of the extended instruction sequences *and* any subsequent sequence IS^{after} is equal or higher than the pET of the sequence IS_{org}^{total} consisting of any of the original instruction sequences *and* IS^{after}.

10.3.1 Definitions

PUB relies on some of the properties brought by time-randomised caches and the way PTA techniques represent the cache state at any given point in the execution of a program. In this section we present those properties

PTA techniques keep a *probabilistic state of the cache* unlike static timing analysis techniques that keep a state of the cache to determine, at any point in the program, which addresses "must", "may" and "won't" be in the cache. We call this state of the cache kept by STA techniques [Wilhelm *et al.* (2008)], *deterministic cache state*.

[1] This option requires controlling the compiler backend passes when the executable code is actually generated.

During the rest of the chapter we assume that the size of the memory unit accessed with a given address $@_k$, usually known as *word*, matches the cache line size. The generalisation to the (actual) case in which a cache line may contain several words is straightforward.

Definition 1. Probabilistic Cache State (PCS). *A given PCS provides the* actual *probability that any given address $@_k$ is present in a time-randomised cache at a given instant t_i of the execution of a program, and thus provides the hit/miss probability for any given address at any point of the execution.*

Definition 2. Probabilistic Execution Time of a Instruction Sequence. *We define the probabilistic execution latency (time) of a sequence of instructions, IS_i, as the execution time distribution it takes the sequence of instructions to be executed: $pET(PCS_{in}, IS_i, PCS_{out})$, where PCS_{in} and PCS_{out} stand for the PCS right before IS_i is executed and the PCS after its execution respectively. Eventually, we also use $pET(PCS_{in}, IS_i)$ if the PCS after the execution of IS_i is irrelevant (e.g., end of the execution of the program), and $pET(IS_i)$ if, additionally, the initial PCS is irrelevant (or provided by the context).*

We say that $pET(IS_i) \geq pET(IS_j)$ if for any cutoff probability the execution time of IS_i is higher or equal than the execution time of IS_j. Figure 10.2 shows the pET of two different sequences IS_i and IS_j where $pET(IS_i) \geq pET(IS_j)$.

Definition 3. Probabilistic Survivability. *We define the probabilistic survivability, or simply survivability, of a given access to an address $@_k$, at any point in the execution of the program as the probability of hit of that access. It is represented as $Surv(@_k)$ and can be approximated by Equation 5.8.*

A cache access may change the PCS of the program that was prior to the execution of such access. This would affect the survivability of all addresses. Given a sequence of cache accesses, the addition of an access A_1 to an address $@_A$, 1) increases or leaves unchanged the survivability of all future accesses A_2 to $@_A$; and 2) decreases or leaves unchanged the survivability of all other addresses. If $@_A$ is in cache when A_1 executes, it does not alter cache contents since with EoM evictions occur only on the event of a miss. If $@_A$ is not in cache, A_1 will fetch it, thus increasing or leaving unchanged the probability of hit of any A_2, though the eviction caused by A_1 can evict other addresses from cache.

10.3.2 Instruction Sequence Subordinance

Definition 4. Instruction Sequence Subordinance. *Starting from an initial PCS, PCS_{in}, a given instruction sequence IS_i is a subordinate instruction Sequence (SIS) of another instruction sequence IS_j if the pET of IS_i is larger or equal than*

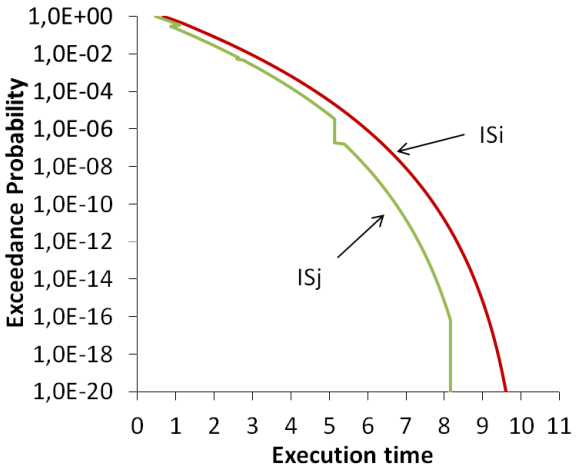


Figure 10.2: Example of comparison of pET

the probabilistic execution time of IS_j. *That is:*$(PCS_{in}, IS_i) \subseteq_{SIS} (PCS_{in}, IS_j) \leftrightarrow pET(PCS_{in}, IS_i, PCS_{outi}) \geq pET(PCS_{in}, IS_j, PCS_{outj})$.

PCS_{outi} and PCS_{outj} are the PCS left after the execution of IS_i and IS_j respectively [1].

Definition 5. Instruction Eviction Upper-Bounding. *An instruction* A_j *is an instruction eviction upper-bounding (IEUB) of* B_j *if the miss probability of* A_j *is higher or equal than the miss probability of* B_j. *This is the case, for instance, if* A_j *is known to be a miss (e.g., it has not been accessed before). Next we deal with the general case where A may have been accessed in the past. Let us consider the following sequences:* $IS_A = <A_i, ..., A_j>$ *and* $IS_B = <B_i, ..., B_j>$ *where* A_i *and* A_j *access address A and no access in between,* $\{X_i\}$, *accesses A; and where* B_i *and* B_j *access address B and no access in between,* $\{Y_i\}$, *accesses B. No relation is established between the addresses in both sequences, so addresses may repeat (e.g., A and B could be the same address).*

A_j *is an IEUB of* B_j, *if we can match all accesses in between* B_i *and* B_j *with accesses in between* A_i *and* A_j *so that in each pair, the access in the sequence* IS_A *is exposed to a larger or equal number of evictions than that of the access in sequence* IS_B. *In this scenario, the miss probability of the access of the pair in* IS_A *is equal or higher than that in* IS_B. *Therefore, the miss probability – and so the pET – of* A_j *is higher or equal than for* B_j.

Note that in a EoM cache, the element determining whether a given access misses in cache or not, is the number of evictions carried out in between that

[1]Note that no cache subordinance, as defined in Section 10.3.3, is established among PCS_{outi} and PCS_{outj}.

access and the previous access to the same address. Further, note that number of evictions that each access suffers depends on the particular initial PCS for each sequence.

As an example let us assume the sequences of accesses $< A_1, B_1, A_2, C_1, B_2>$ and $<D_1, E_1, F_1, D_2, E_2>$ and an empty initial cache state, with X_i accessing address X. In this case D_2 is a IEUB of A_2. This is so because we can pair up B_1 with, for instance, E_1 (both are misses). Similarly, E_2 is a IEUB of B_2 since we can pair up C_1 with F_1 (both are misses) and A_2 with D_2 since D_2 has to survive to 2 evictions whereas A_2 has to survive to 1 eviction.

Definition 6. Sequence Eviction Upper-Bounding. *IS_i is a sequence eviction upper-bounding (SEUB) of sequence IS_j if we can match all accesses in IS_j with accesses in IS_i so that in each pair the access in IS_i is a IEUB of the access in IS_j. Then, the pET of IS_i is higher or equal than the pET of IS_j. This makes IS_i a SIS of IS_j.*

SEUB definition is particularly useful to compare sequences whose initial PCS is the same as shown later. The definition of SEUB is related to a similar observation done in the context of probabilistic time composability, which is discussed in the next chapter.

Theorem 1. Starting from an initial cache state PCS_{in}, the execution time of the instruction sequence IS_{orig} increases if one access, C_1, to any address C – regardless of whether there are other accesses to C in IS_{orig} –, is introduced at any point of this sequence, thus obtaining $IS_{ext} = IS_{orig} \cup C_1$. That is, $(PCS_{in}, IS_{ext}) \subseteq_{SIS} (PCS_{in}, IS_{orig})$. This is so because IS_{ext} is a SEUB of IS_{orig}.

The proof to this Theorem is provided in Section 10.4.

Applicability to PUB: The fact that adding cache accesses increases the pET of a sequence allows us to add accesses in the different branches of the CFC until all branches in the extended code include all sequences of accesses in all branches in the original code. Therefore, any branch in the new code will be a SIS of all branches in the original code. This property helps understanding the *PUB* variant called *PUBam* (Section 10.5.1).

10.3.3 Cache Subordinance

Definition 7. Unique Address. *A given cache access is called unique or said to access a unique address $@_{un}$ if it can be ensured that its survivability is zero. This happens when $@_{un}$ is accessed for the first time or it can be ensured that it was evicted since last time it was accessed, because an invalidation instruction is executed on that address after its last access. For the rest of the discussion, it*

is assumed that unique addresses are evicted right after they is accessed, so its survivability is always zero.

Definition 8. Probabilistic Address Subordinance. *A given PCS_i subordinates another PCS_j for a given access to address $@_k$ if the probability of hit of that access, i.e. its survivability, is lower or equal in PCS_i than in PCS_j. That is, $PCS_i(@_k) \subseteq PCS_j(@_k) \leftrightarrow Surv(PCS_i, @_k) \leq Surv(PCS_j, @_k)$.*

Definition 9. Probabilistic Cache Subordinance. *A given PCS_i subordinates another PCS_j if for any address, $@_k$, the survivability of an access to $@_k$ is lower or equal in PCS_i than in PCS_j, which is represented as follows:*

$PCS_i \subseteq PCS_j \leftrightarrow \forall @_k : PCS_i(@_k) \subseteq PCS_j(@_k)$.

If $PCS_i \subseteq PCS_j$, and assuming that core operations have a fixed impact on pET, then the pET for any sequence $IS_i = \{I_1, I_2, ..., I_n\}$ is higher when the initial cache state for IS_i is PCS_i than when it is PCS_j. This means that for any exceedance probability, the execution time is higher or equal when the initial cache state is PCS_i than when it is PCS_j. This is so because under PCS_i every single access has a lower hit probability than under PCS_j, which translates into a equal or higher pET when starting with PCS_i.

Let us consider the sequence of accesses to memory $IS_{org} = <I_1, I_2, ..., I_n>$. In this sequence there is no constraint on whether each $I_i (i \in [1..n])$ accesses a different cache line or one already accessed by another I_j, or whether data were in cache before.

Theorem 2. Let us assume a given PCS_{org} and the instruction sequence, IS_{org}, whose execution moves the cache state to $PCS_{org-out}$. Further assume that we replace an access I_i by an access, C_1, to a unique address C in any position in IS_{org}, so that we obtain $IS_{ext} =< I_1, I_2, ..., I_{i-1}, C_1, I_{i+1}, ..., I_n >$. In this scenario: (1) The pET of IS_{ext} is higher or equal to that of IS_{org}, that is, $pET(IS_{ext}) \geq pET(IS_{org})$, and (2) the PCS after executing IS_{ext}, $PCS_{ext-out}$, is a subordinate of $PCS_{org-out}$, i.e. $PCS_{ext-out} \subseteq PCS_{org-out}$ for all addresses.

This theorem talks about the fact that if we replace any access in a sequence by an access to a unique address (especial address that cannot be in cache before it is accessed and it is evicted immediately after being fetched so that further accesses to C cannot hit in cache), the pET of the IS_{ext} and any subsequent sequence IS_l is higher than the pET of IS_{org} and IS_l. That is, $pET(IS_{ext} + IS_l) \geq pET(IS_{org} + IS_l)$. This property helps understanding the *PUB* variant called *PUBaa* (Section 10.5.2).

10.3.4 Theorem 1 in Time-Deterministic Caches

In time-deterministic caches Theorem 1 does not apply. For instance let us assume a fully-associative two-entry cache deploying LRU replacement. Further assume the sequence $IS_{org} = <A_1, B_1, C_1, A_2>$, where accesses A_i access address A, accesses B_i access address B and accesses C_i access address C. Further A, B and C correspond to different addresses (mapped to different cache lines). That sequence experiences exactly 4 misses when accessing cache. If we add one access to address A after B_1, so $IS_{ext} = <A_1, B_1, A_3, C_1, A_2>$, the new sequence experiences only 3 misses (A_1, B_1 and C_1) despite the insertion of an extra access since A_3 and A_2 would hit in cache. As a result, the execution time of IS_{ext} is shorter than the execution time of IS_{org}, while the cache state left is the same (A and C, with C being the LRU element).

10.4 Proof for Theorem 1 and Theorem 2

Proof Theorem 1. The new access to address C (C_1) is introduced in a particular location in the sequence. If C_1 hits in cache, each instruction in IS_{ext} is a IEUB of its counterpart instruction in IS_{orig}, that is, each instruction is subject to the same number of evictions in both sequences. As a result, IS_{ext} is an SEUB of IS_{orig}. In essence, the pET of IS_{ext} increases by the latency of a cache hit w.r.t IS_{orig}.

If C_1 misses in IS_{ext} and C is never accessed again, all instructions in IS_{ext} are an IEUB of their counterpart instruction in IS_{orig} and hence IS_{ext} is a SEUB of IS_{orig}. In this case, the pET of IS_{ext} increases by the latency of a cache miss w.r.t IS_{orig}. This case would correspond to that in Theorem 2.

If there is an access to C, C_2, after C_1, given that C_1 misses in IS_{ext} and it occurs before C_2 (Figure 10.3a), C_2 is effectively a miss in IS_{orig}. C_1 in IS_{ext} is an IEUB of C_2 in IS_{orig} since both are misses. However, C_2 in IS_{ext} may become a hit in cache, thus reducing the number of evictions suffered by the accesses to any address after C_2. In other words, sequences like this $IS_{orig} = <A_1, ..., C_2, ..., A_2>$, where A_i and C_i access different addresses and in between them there can be any sequence of accesses to addresses other than A and C, may observe some gains, i.e. reduction in pET, for A_2 when C_1 is added before C_2. Note that we focus on the case where A_2 misses in IS_{org} and potentially hit in IS_{ext} (if A_2 were a hit in IS_{org} the inclusion of C_1 would not reduce its latency any further). Despite that, it is still the case that, IS_{ext} is a SEUB of IS_{orig}. In both sequences we ignore the effect on the other accesses other than A_1 and A_2. Once the whole analysis is complete, it can be extended to any other access. We identify the following scenarios:

- If C_1, which misses in cache, is introduced between the two accesses to A, leading to the instruction sequence $IS_{ext} = <A_1, ..., C_1, ..., C_2, ..., A_2>$ and this may make C_2 become a hit, still A_2 in IS_{ext} will suffer as many evictions in IS_{ext} as in IS_{orig} – at least one in the former and exactly one in the latter, see Figure 10.3a. Hence A_2 in IS_{ext} is a IEUB A_2 in IS_{orig}. As before C_1 in the extended sequence is a IEUB of C_2 in the original.

- If C_1 is added before A_1, hence leading to $IS_{ext} = <C_1, ..., A_1, ..., C_2, ..., A_2>$, we identify two cases: (1) A_2 misses in IS_{orig}, because it is evicted by an access B_1, with B_1 and C_2 accessing different addresses; and (2) and no access B_1 evicts A_2 and C_2 may evict A_2.

  - Under (1) whether C_2 becomes a hit due to the inclusion of C_1 has no effect on A_2. As a result, C_1 and A_1 in IS_{ext} are IEUB of C_2 and A_1 in IS_{orig}, while A_2 remains being a miss (see Figure 10.3b).
  - Under (2), we identify two final sub-scenarios: (2.a) A_1 is a miss in IS_{orig} and (2.b) it is a hit.

    Under (2.a), we can match C_1 and A_1 of IS_{ext} with C_2 and A_1 of IS_{orig}, since all of them are misses. We can also match C_2 of IS_{ext} with A_2 of IS_{orig}, as they suffer one eviction — C_2 suffers the eviction caused by A_1 in IS_{ext} and A_2 suffers the eviction caused by C_2 in IS_{orig} (Figure 10.3c).

    Under (2.b), we can match C_1, A_1 and C_2 of IS_{ext} with C_2, A_2 and A_1 of IS_{orig} respectively as C_1 (IS_{ext}) and C_2 (IS_{orig}) miss both, A_1 (IS_{ext}) and A_2 (IS_{orig}) suffer one eviction each (by C_1 and C_2 respectively), and C_2 (IS_{ext}) can be a hit in the best case and A_1 (IS_{orig}) is a hit (Figure 10.3d).

In summary, in all cases adding an access C_1 to the original sequence IS_{orig} makes that the new sequence, IS_{ext}, becomes a SEUB of IS_{orig}, and so have a higher pET. Hence, IS_{ext}, where $IS_{ext} = IS_{orig} \cup C_1$, is a SIS of IS_{orig} as $pET(PCS_{in}, IS_{ext}) \geq pET(PCS_{in}, IS_{orig})$. □

We further illustrate this with an example of a complex sequence. Let us assume $IS_{orig} = <A_1, B_1, C_1, D_2, B_2, C_2, A_2>$ and $IS_{ext} = <A_1, D_1, B_1, C_1, D_2, B_2, C_2, A_2>$ after adding D_1. The initial PCS is empty. We can match the following pairs from $[IS_{ext}, IS_{orig}]$: $[A_1, A_1]$, $[B_1, B_1]$, $[C_1, C_1]$, $[D_1, D_2]$ as all of them are misses; $[D_2, B_2]$ as they suffer both 2 evictions (and they generate δ evictions); $[B_2, C_2]$ as they suffer $1 + \delta$ evictions (and generate γ evictions); and $[A_2, A_2]$ as they suffer $3 + \gamma$ evictions in IS_{orig} and $3 + \gamma + \theta$ evictions in IS_{ext} where θ is the number of evictions produced by C_2 in IS_{ext}. Overall, IS_{ext} is a SEUB of IS_{orig}.

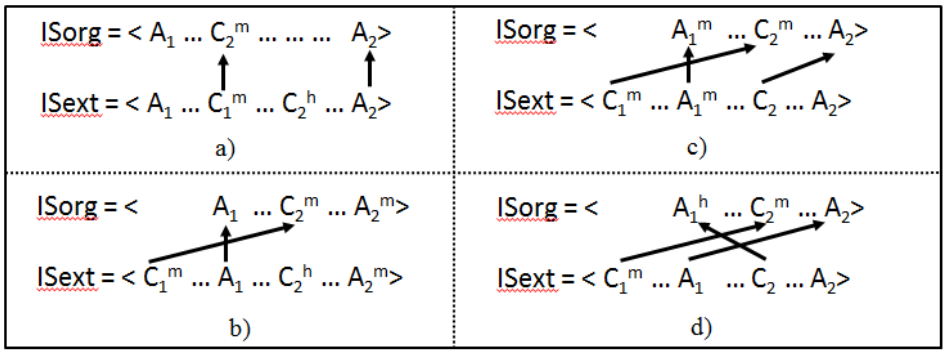


Figure 10.3: Illustration of possible cases in Proof 1. For each access, e.g A_1, the superindex indicates whether it is a hit A_1^m or a miss A_1^m. Arrows shown which access in each sequence can be paired up according to IEUB definition

Proof Theorem 2. Obviously, the survivability of the accesses executed before C_1 is not affected by C_1. Since C is not present in cache, access C_1 introduces *exactly* one eviction in a random line in cache – with each line having $\frac{1}{W}$ probability to be evicted. This decreases, or does not affect, the survivability of subsequent accesses. The assumption that C is evicted after it is accessed makes that no future access to C benefits from it. Further, the execution of C (miss latency) is equal or higher to that of the access it replaces, thus increasing (or leaving unmodified) the pET of the extended sequence, making $pET(IS_{ext}) \geq pET(IS_{org})$.

□

10.5 Applying PUB

Based on Theorems 1 and 2, next we show how *PUB* extends programs so that they upper-bound the probabilistic timing behaviour of their original counterparts.

CFC increase the complexity of the WCET estimation using MBPTA. In order for *PUB* to produce an upper-bound of the pWCET of CFCs, requirement *PUBReq1* or requirement *PUBReq2* should hold:

PUBReq1. *PUB* has to derive a bound, $pETB$, to the longest pET of the execution of any of the i branches, IS_{bb_i}, in the CFC plus any instruction sequence IS_{after} after the CFC. Note that the pET of IS_{after} is affected by the particular branch that is executed in the CFC. Overall, *PUBReq1* imposes that $pETB \geq pET(IS_{bb_i}) + pET(IS_{after}), \forall bb_i$.

PUBReq2. *PUB* has to upper-bound the longest pET of all the branches in the CFC and the worst probabilistic cache state left by any of the branches of the CFC. If this is achieved for every CFC, the probabilistic execution time of any path in the extended code is higher than the execution time of any path in the original code.

To reach its goal *PUB* deploys different solutions for the core latency and for cache (either data or instruction) latency. We focus on cache latency next and we address core latency in Section 10.5.4. We start with the application of *PUB* to the data cache. Its extension to the instruction cache is straightforward and it is covered in Section 10.6.

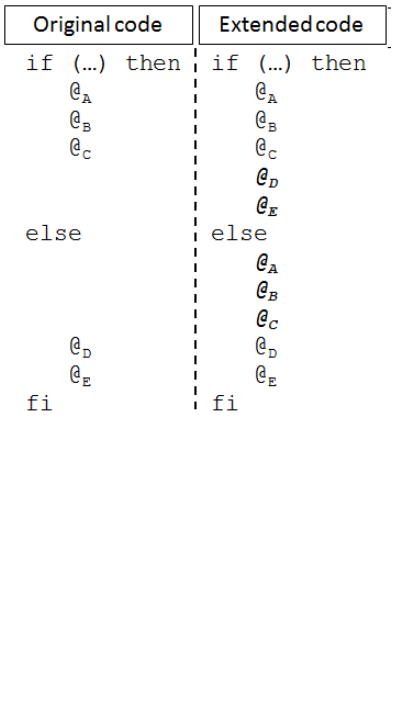
The following example illustrates the principle of *PUB* for the data cache. *Example 1:* Let us assume an `if-then-else` construct prior to which the cache is in a given PCS, PCS_{in}. The purpose of *PUB* is to extend both branches of the conditional statement, IS_{org}^{left} and IS_{org}^{right}, into IS_{ext}^{left} and IS_{ext}^{right} such that the impact in the pET of the whole program of both branches in the extended code, IS_{ext}^{left} and IS_{ext}^{right}, upper-bounds the pET of the program for both branches in the original code, IS_{org}^{left} and IS_{org}^{right}.

We propose two different alternatives *Address Merging* (*PUBam*) and *Address Aging* (*PUBaa*) respectively.

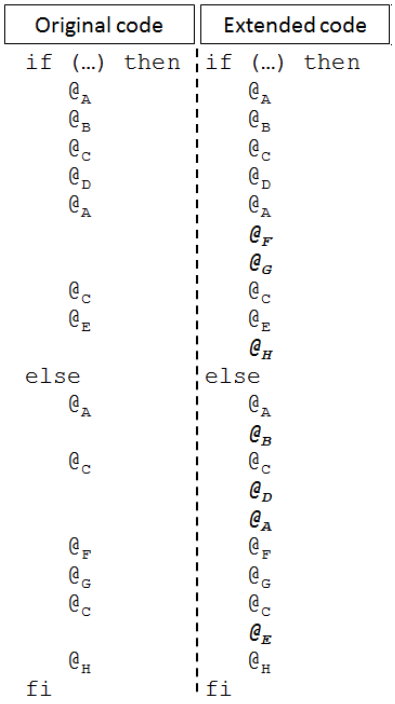
10.5.1 Address Merging (*PUBam*)

PUBam makes the different branches of the extended CFC perform at least the same data accesses (same addresses) and in the same order as any of the branches of such CFC in the original code. For instance, let us assume an `if-then-else` construct as shown in Figure 10.4(a) where addresses $@_A$, $@_B$ and $@_C$ are accessed in the "if" branch and $@_D$ and $@_E$ in the "else" branch. In this case, *PUB* has to ensure that accesses $@_A$, $@_B$, $@_C$, $@_D$ and $@_E$ occur in both branches, making sure that the relative order of $@_A$, $@_B$ and $@_C$ is maintained, and so it is also for $@_D$ and $@_E$. Thus, for instance, we could access $<@_A, @_B, @_C, @_D, @_E>$, or $<@_D, @_E, @_A, @_B, @_C>$, or $<@_A, @_D, @_B, @_E, @_C>$, but not $<@_A, @_E, @_D, @_C, @_B>$.

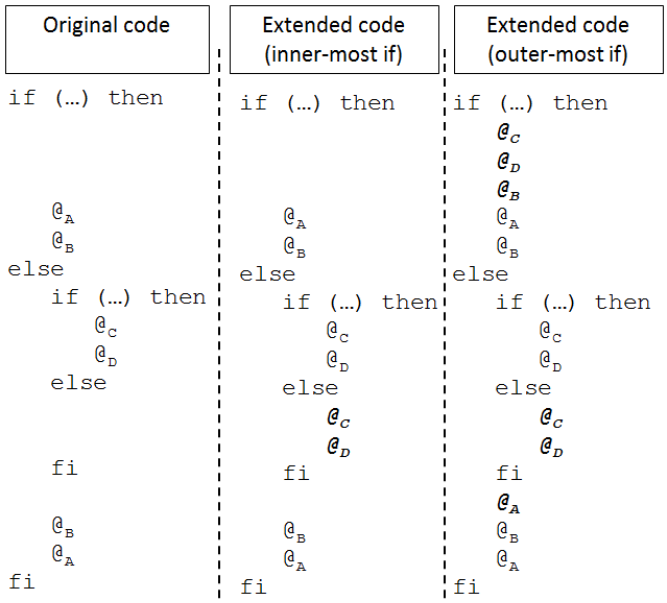
While the solution based on replicating all accesses in all branches of the CFC can always be used, it can be optimised by avoiding unnecessary replication of accesses. For that purpose, we identify the longest sequence of accesses that occurs to the same addresses and in the same order (although not necessarily consecutively) in all branches, and do not repeat them. Then, those accesses in the other branch excluding the repeated ones must be interleaved with the ones in the current branch in a way that the relative order of the accesses in the other branches is maintained. This is better illustrated with an example. Figure 10.4(b) shows an example of an `if-then-else` construct with a sequence of accesses that repeats in both branches, and how the code could be modified to upper-bound both branches regardless of the branch taken. As shown in the figure, a sequence with 3 accesses in the same order occur in both branches of the `if-then-else` construct: $<@_A, @_C, @_C>$. Branches are upper-bounded by properly placing accesses $@_F$, $@_G$ and $@_H$ in the "if" branch, and $@_B$, $@_D$, $@_A$ and $@_E$ in the "else" branch.



(a) Simple Code Replication



(b) Max. Common Pattern Identification and Code Replication



(c) Code Replication for a nested *if*

Figure 10.4: Examples of data cache branch upper-bounding

Different ways to interleave branches are also valid. For instance, $<@_D, @_A>$ and $<@_F, @_G>$ could be swapped. In any case, the new sequence of accesses upper-bounds the execution time of the original code in any of both branches in terms of cache behaviour.

Link to Theorem 1: Let us denote the initial cache state before the CFC as PCS_{in}, the sequence of (memory) instructions in any of the branches as IS_i and the one after the CFC as IS_{after}. We observe that:

- Adding to IS_i a data access present in any of the other branches of the CFC, leads to the extended sequence IS_j. According to Theorem 1, $(PCS_{in}, IS_j \cup IS_{after}) \subseteq_{SIS} (PCS_{in}, IS_i \cup IS_{after})$, so the total pET of $IS_j \cup IS_{after}$ increases when IS_j executes instead of IS_i since $IS_j \cup IS_{after}$ is a *SIS* (subordinate instruction sequence) of $IS_i \cup IS_{after}$.

- We can repeat this process incrementally adding those accesses in the other branches of the CFC, one by one, keeping the same relative order of those accesses in their original branches. Every time we add a new data access, the resulting sequence is a *SIS* of all previous sequences, and so of the original branch IS_i.

- Given that for each branch, b, we add all the accesses that are in the other branches but not in b, also makes the extended version of each branch be a SIS of all original branches.

This process can be carried out using a bottom-up approach starting from the innermost CFCs until the whole program is analysed adding the corresponding data accesses. By doing so *PUBReq1* is met.

Different CFC. So far we have considered simple `if-then-else` constructs. Next, we consider other types of CFC such as `if-then`, `switch`, `loop` and nested `if-then-else` constructs.

`If-then`. An if-then construct can be treated analogously to `if-then-else` ones assuming that the "else" branch is empty, so that the accesses in the "if" branch are simply added in the new "else" branch.

`Switch`. In switch (or `if-then-elsif-elsif-...`) constructs, the number of branches can be larger than 2. The sequence of repeated accesses should, therefore, exist in *all* branches. All accesses in *all* the remaining branches excluding the repeated sequence (repeated in *all* branches) are placed appropriately by *PUB* in the current branch.

`Nested Conditionals`. Regarding nested CFC, our mechanism can be applied recursively starting from the innermost CFC. When moving one level up, since all the branches of the innermost CFC are trustworthy upper-bounds of the original branches, any of them can be used. When a number of instructions is to be

inserted in one branch of the outer CFC, they must be inserted identically in all branches of all inner CFC. An example of nested CFC, with `if-then-else` and `if-then` constructs is shown in Figure 10.4(c). The first column shows the original code. The second column the branch upper-bounding for the innermost if-then that becomes an `if-then-else`. Finally, the last column shows the code after branch upper-bounding for the outermost CFC. As shown, the "if" branch of the outermost if-then-else construct must be also extended with the accesses in the if-then-else construct inside the "else" branch.

`Loops`. Branch upper-bounding is trustworthy even in the case of loops because although data cache accesses may change across iterations (e.g. vector traversals), accesses in any of the branches of the CFC remain the same. Note that not replicated accesses are only those that can be guaranteed to access the same address in all branches of the CFC *in every iteration*. For instance, if two accesses occur to a position in an array in the different branches of a CFC and they are guaranteed to access the same position (e.g., $a[i]$ where i is the loop index), then the mechanism is trustworthy. If those accesses cannot be proven to access the same position in all iterations (e.g., $a[i]$ and $a[j]$), then they are assumed not to access the same address and are replicated in the different branches of the CFC.

`Infeasible paths, error codes and modes of operation`. Unless instructed otherwise *PUBam* balances the different branches on every CFC. However, there are several circumstances in which one or several of the branches of a CFC do not need to be considered in the computation of the pWCET, or at least do not have to be considered together. For example, if the user deems some CFC as non-relevant for the WCET because their execution does not affect the WCET (e.g., code to deal with error conditions), the user can instruct *PUB*, e.g., by means of annotations, not to balance instructions in those branches, thus reducing *PUB* overhead.

Operation modes is another example in which the user can help *PUB*. Software with different operation modes, which can be mutually exclusive, encapsulates different functionalities for each of which a different pWCET can be derived to reduce the pessimism that an across-modes pWCET would incur. In order to prevent *PUB* from generating a pWCET that upper-bounds all those together, the user must correctly identify them and indicate to *PUB* the code belonging to different operation modes. Then the user can provide path coverage for those operation modes (i.e. one input per mode). For instance, in a switch statement in which each case represents a different mode, it would be simple to annotate the code to prevent *PUB* from balancing the different branches of the switch.

`Function Calls.` If a function is called in one branch of a CFC, its effect on the cache has to be replicated on the other branches. This can be done either (1) by creating a dummy function that accesses the same addresses in the same order,

or (2) by means of the Address Aging technique presented next. Further, if the function is called in *all* branches of a CFC with the same inputs then the function can be considered the 'common pattern' in all branches of the CFC, applying *PUB* only to the code before the call and after the call to the function.

10.5.2 Address Aging (*PUBaa*)

The fact that *PUBam* requires merging accesses of all branches of a CFC may lead to pessimistic pWCET estimates, which we call *PUBam* inefficiency. This pessimism manifests itself when the code in any branch of a CFC in the extended version has significantly higher number of accesses than that in any branch of the original version. This results in a PCS after the execution of the *extended version* of any of the branches of a CFC that is worse (i.e. the hit probabilities of any subsequent accesses is lower) than the worst PCS obtained after executing any of the branches in the *original CFC*.

PUBam introduces low overhead on `if-then-else` CFCs in which in one of the branches b_{1-org} few accesses to cache are carried out. In that scenario the addresses of the other branch b_{2-org} are copied into b_{1-org} leading to b_{1-ext}, with b_{2-ext} almost unchanged with respect to b_{2-org} since b_{1-org} contains few accesses. In this case the worst cache behaviour in the extended version, which happens when b_{2-ext} executes, is almost the same as when b_{2-org} executes. Similarly, *PUBam* introduces little overhead if both CFC branches perform similar access sequences as few extra accesses need to be added. Note that *PUBam* also handles efficiently `if-then` constructs, which is the case when one of the branches is empty as *PUBam* keeps the worst path untouched and only increases the code in the missing path.

PUBam is inefficient when in each branch of a CFC different sets of addresses are accessed. In that case, all branches have to be extended adding the accesses present in the others, making that the execution time and the PCS after executing any of the extended branches are much worse than those obtained after executing any of the original ones. For instance, if we have a 10-case switch such that in each of its branches 2 different addresses are accessed, this would result in an extended switch containing 20 accesses in each branch.

In order to handle this inefficiency of *PUBam* we propose a new technique called address aging (or *PUBaa*). Instead of copying addresses of one branch on the rest, *PUBaa* adds accesses to addresses accessed nowhere else in the program, which lead to a miss and fetch no useful data. In particular, it adds as many accesses as the maximum number of accesses in any of the branches. In the previous example of a 10-case switch in which each branch has 2 different accesses, in each branch *PUBaa* adds 2 accesses to unique addresses.

Link to Theorems 2 and 1: The idea is that in the original switch the worst situation happens when the branch executed incurs two misses. If we consider only

the 2 added accesses which miss and fetch no useful data, then their execution time and the PCS they leave are worse than those in the original code in all branches. If we consider that those two accesses virtually replace the accesses in the other branches of the original program, Theorem 2 applies, meeting *PUBReq2*. The fact that in the branch the two missing accesses are added back (those in the original code) only increases the execution time according to Theorem 1, hence meeting *PUBReq1*.

Note that *PUBaa* is particularly good when the number of branches in the CFC is high. Conversely, if the number of branches is low and they potentially present a high imbalance, *PUBam* is more efficient. This makes *PUBam* and *PUBaa* complement each other.

PUBaa can be implemented with or without hardware support. If no hardware support is in place, accesses added by *PUBaa* are ensured to miss by accessing addresses never accessed before. A data structure (*dummy*) and a pointer (*next*) are created so that the address accessed by any of those accesses is the one pointed by *next* (so *dummy*[*next*]) and *next* is increased by the cache line size of the affected caches. Alternatively, *PUBaa* can be implemented with hardware support by adding a special instruction, $MissPubaa$ that creates a miss in cache, hence creating a random eviction, and accesses memory, but that brings no useful content to cache.

10.5.3 Creating the *PUB* Code

Adding memory operations can be done in several ways depending on the hardware characteristics. If a non-modifiable register exists (e.g., register $r0$ in SPARC and MIPS architectures, which is hardwired to zero), new accesses can be translated into `r0 = LOAD` $@_A$, where $@_A$ is the particular address to be accessed. If such a register does not exist, then a free register must be used to hold the data read from memory. Such data will not be used, but the semantics of the instruction set architecture (ISA) must be respected when reading data from memory. Finally, note that in some cases accessing an address $@_A$ that was not accessed in the original program may create an exception. This is not often the case since most memory operations can be guaranteed not to access beyond the memory bounds of the program. For the remaining accesses, if *PUBaa* is applied, as mentioned before, it is required creating a data structure so that unique accesses to that structure age the cache state appropriately if no hardware support is in place.

Code Alignment. When applying *PUB*, code (instruction accesses) may get unaligned with respect to the original code, thus altering the timing behaviour in a way that may end up not upper-bounding the original code. It is important realising that the survivability of a given access, i.e. its hit probability, does not depend on the particular address it is mapped in memory. The address in

a deterministic cache would determine the particular set in which the address is mapped in cache, but this is not the case in a time-randomised cache, since the placement is randomised breaking the dependence between the address of an access and its assigned cache set. Given two accesses to cache what really affects their hit/miss probability is whether they are mapped to the same cache line or not, which in turn affects their reuse distance (represented as k in Equation 5.8). Hence, a principle we follow when applying *PUB* is not to unalign basic block boundaries, so that, instructions that in the original code are mapped to the same line out of the basic block being modified remain mapped to the same line in the extended code and vice-versa. Also, the code in the basic block – whose all instructions are always traversed sequentially – keeps the same cache-line alignment at its boundaries so that it has the same behaviour as is the original code increased with the access to some extra cache lines. This is achieved by ensuring that *PUB*-added instructions are added so that the total size of the code added in each of the branches of the CFC is a *multiple of the cache line size*. This makes that the rest of the code keeps *exactly* the same cache-line alignment as in the original code. Again, note that altering the actual addresses of the pieces of code is irrelevant given that MBPTA-compliant designs such the ones proposed in the Chapter 4 or this thesis are built upon random placement caches or software randomisation is used on top of deterministic placement (e.g., modulo placement) caches.

10.5.4 Core Latency

We focus on a processor architecture deploying time-randomised instruction and data caches with a core pipeline similar to the LEON4 [Cobham Gaisler (2011)] processor in which processor instructions have a fixed latency, as it has been shown to be PTA-compliant [Kosmidis *et al.* (2013a)]. Core operations, besides their access to the instruction cache, which is covered in Section 10.6, do not affect the PCS of the data cache.

PUB computes the core latency of the instructions in the different branches of a CFC and adds core instructions to each branch such that the core latency of all branches is equalised[1]. *PUB* adds the minimum number of instructions taking into account the timing effect that they introduce. Added instructions will be typically arithmetic-logic operations, such as integer or floating point additions, multiplications, etc. that have a neutral effect on the functional behaviour of the program by either writing the same value read (e.g., multiplying by 1) or writing into a hardwired constant-value register.

[1] In this step memory operations (loads and stores) are ignored.

10.5.5 Steps

PUB carries out several steps that should be applied in the following order: First, core latency and data cache latency are upper-bounded in any order. Then, instruction cache latency is upper-bounded (excluding code alignment) for efficiency reasons, since it can make use of those instructions already added by the previous steps. Finally, code alignment is performed.

10.6 PUB for Instruction Caches

The same principles applied to the data cache apply to the instruction cache, although the instruction cache has several peculiarities. Three main aspects are considered by *PUB* to deal with the instruction cache: code invocation, function calls and code alignment.

Code Invocation. For the sake of this explanation let us assume an `if-then-else` construct where IS_{left} is executed in the "then" branch and IS_{right} in the "else" branch. We would like to apply the same solution as for data, where accesses in other branches of a CFC are reproduced in the current one. Ideally, this could be conceptually achieved if we could access all the instruction addresses of the CFC, every time any branch of the CFC is executed. In the `if-then-else`, if we were able to create $IS_{join} = IS_{left} \cup IS_{right}$, then IS_{join} would be a SIS for both IS_{left} and IS_{right}, or formally stated:

$$(PCS_{in}, IS_{join}) \subseteq_{SIS} (PCS_{in}, IS_{left})$$
$$(PCS_{in}, IS_{join}) \subseteq_{SIS} (PCS_{in}, IS_{right})$$

Then, we would like to have IS_{join} in both branches of the `if-then-else` construct so that the whole `if-then-else` construct in the extended version $CFC_{ideal} =< if\ IS_{join}\ else\ IS_{join} >$ is a SIS of the whole `if-then-else` construct in the original version $CFC_{org} =< if\ IS_{left}\ else\ IS_{right} >$

However, this is not possible because the only way to access an instruction address is to fetch and execute the instruction placed in this memory location. In the case of the CFC this requires to modify the control flow of the execution such that both branches of the CFC are executed always, which would have undesirable functional effects. Since the particular addresses are irrelevant for time randomised caches, we can produce the same timing effect by building $CFC_{real} =< if\ IS_{join-left}\ else\ IS_{join-right} >$ where $IS_{join-left} = IS_{left} \cup IS_{right'}$ and $IS_{join-right} = IS_{right} \cup IS_{left'}$, such that $IS_{left'}$ and $IS_{right'}$ have the same pattern of access to the instruction cache as IS_{left} and IS_{right} respectively. Later we explain how to make $IS_{left'}$ and $IS_{right'}$ not to have any functional effect.

If CFC_{real} is executed just once, its timing behaviour will be exactly the same as for CFC_{ideal}.

If the CFC_{ideal} is executed several times (which is the case in a loop), being IS_i, IS_j, ... the sequences of instruction executed after every time CFC_{ideal} runs, then we would have the following sequence of execution: $IS_{join}, IS_i, IS_{join}, IS_j, IS_{join}, IS_k, IS_{join}, ...$

With CFC_{real}, any of the branches $IS_{join-left}$ or $IS_{join-right}$ can be executed instead of IS_{join}. Hence, if in the different runs of the CFC, the same branch is always executed, the reuse distances and so the hit/miss probabilities – and so the pET – will be the same as in the ideal case where IS_{join} is executed always. However, if different branches are executed as, for instance, in the following sequence: $IS_{join-left}$,IS_i,$IS_{join-right}$,IS_j,$IS_{join-right}$,IS_k,$IS_{join-left}$,... then, reuse distances grow w.r.t. IS_{join}. As a consequence, any sequence of paths of $if\ IS_{join-left}\ else\ IS_{join-right}$ experiences the same execution patterns as $if\ IS_{join}\ else\ IS_{join}$, but with equal or higher reuse distances, and thus with equal or higher pET. Therefore, $if\ IS_{join-left}\ else\ IS_{join-right}$ is a SIS of $if\ IS_{join}\ else\ IS_{join}$, which is already a SIS of $if\ IS_{left}\ else\ IS_{right}$.

So far we have assumed that $IS_{left'}$ and $IS_{right'}$ do not have any effect on the functional behaviour of the program. This challenge can also be addressed easily by making sure that the added instructions do not impact the functional behaviour — for example making write operations access addresses not used by the program. In the simplest case, if the code to be replicated from other branches is sequential (no CFCs or loops), one can simply add as many *nop*[1] as needed so that the code size of any branch of the conditional construct has the total size of adding all code in all branches. *PUB* therefore adds some instructions in the different branches of the CFC. However, since data cache and core instructions branch upper-bounding may have already added some instructions, we only need to add the remaining ones (if any).

The solution described can be applied as it is in the context of *PUBam*. In the case of *PUBaa* one cannot easily introduce instructions that always miss in cache without hardware support. However, the *MissPubaa* instruction described before can be made also pay an instruction miss. For this purpose, whenever the instruction is fetched (which may hit in the instruction cache), instead of fetching the next one, an instruction miss operation is started. Whenever complete, a cache line is evicted from the cache and fetch resumes.

Function Calls. The same solution used for data cache branch upper-bounding solves the issue for the instruction cache. If a function is called from all branches of the conditional CFC with the same inputs, they will exercise the same code on the instruction cache – note that all branches of the function will be also upper-

[1] A *nop*, no-operation, instruction is any instruction producing no functional effect in the execution of the program. In some ISAs such an instruction exists. If this is not the case, it is always possible performing operations like adding "0" to any particular register.

bounded so it will be irrelevant the actual branches exercised in each invocation of the function.

Code Alignment. Code alignment issues have been already explained in Section 10.5.3. Note, however, that code alignment must be the last step in the process.

10.7 Evaluation

We use the experimental setup (simulator and benchmarks) described in Chapter 3.1.2. Our architecture ensures that requests sent to each resource are issued in program order and are also served in program order to prevent timing anomalies by construction [Wenzel *et al.* (2005a)].

The size of both the I-cache and the D-cache is 8-KB with 16-byte lines and 8-way set-associativity. Both caches, instruction and data, implement random placement and replacement policies. The D-cache is write back. On a dirty line eviction the pipeline is stalled. In our architecture stores may have lower impact on execution time than loads due to the use of a store buffer. To handle this PUB added memory operations are loads.

The latency of the fetch stage depends on whether the access hits in I-cache (1 cycle) or misses (100 cycles). After the decode stage, memory operations access the D-cache whose behaviour is analogous to that of I-cache.

We use benchmarks from the EEMBC Autobench [Poovey (2007)] and Mälardarlen [Gustafsson *et al.* (2010)] suites as reference for the analysis. EEMBC Autobench is a well-known benchmark suite that reflects the current real-world demand of some embedded systems. Mälardarlen benchmarks [Gustafsson *et al.* (2010)] are also commonly used in the community to evaluate and compare different types of WCET analysis tools and methods.

For several of these benchmarks we have the input set leading to the worst-case path. For some of them, the number of iterations they carry out depends on the input data. The input data we have for them provide the worst-case path for the a given loop-iteration count that we assume fixed. The benchmarks in this group are: `bs`, `cnt`, `fir`, `lcdnum`, `prime`, `insersort`, `edn`, `matmult`, `janne`, `cover`, `fdct`, `fibcall`, `jfdctint`. For these benchmarks, the ratio between the WCET estimation with PUB and the one with MBPTA corresponds to the actual WCET overestimation introduced by PUB. For the rest of the benchmarks, for which we could not derive the worst-case path, PUB WCET overestimation is an upper-bound of its actual WCET overestimation, since another input set could exist that exercises the worst path, making MBPTA pWCET estimate to increase and be closer to the one provided by PUB.

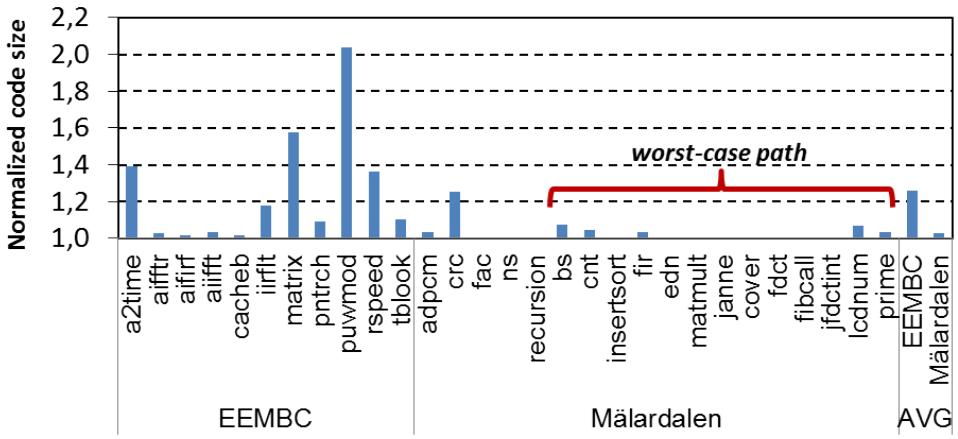


Figure 10.5: Impact on code size of *PUB*.

Note that the effort and time to implement PUB in a compiler framework exceeds by far what is feasible for a thesis. For this reason we opted to apply PUB manually in the source code, by adding data references in each path. This way we mimic its overheads and gain some insight on the cost of the technique.

10.7.1 Code Replication Size

It is important to note that with *PUB the extended program can be used only at analysis time to derive pWCET estimates, while at deploy time the original program can be used*.

Figure 10.5 shows the code-size overhead introduced by *PUB*. We observe that for most of the benchmarks the overhead is below 20%, with only few benchmarks introducing higher overhead. The case of few EEMBC benchmarks (e.g., *puwmod*, *matrix* and *a2time*) is remarkable since the code size is increased noticeably. In contrast, some Mälardarlen did not require any change (e.g., *cover*, *edn*, *fdct*, *fibcall*, *insertsort*, *janne*, *jfdctint*, *matmult* and *ns*) since they do not have any conditional CFC or those conditional CFCs exist but they are already fully balanced (e.g., *cover* has some constructs where a given variable is incremented or decremented in different branches but cache is not accessed).

Regarding those benchmarks experiencing a significant code size increase due to *PUB*, we have observed that this highly correlates with the number of conditional CFCs and their degree of nesting. For instance, the four EEMBC benchmarks with lowest number of conditional CFCs (up to 6) are *aifftr*, *aifirf*, *aiifft* and *cacheb*, and are the ones experiencing the lowest code increase. Conversely, *puwmod* is the benchmark with highest number of conditional CFCs (66) and frequent conditional CFC nesting (often 3 conditional CFCs nested).

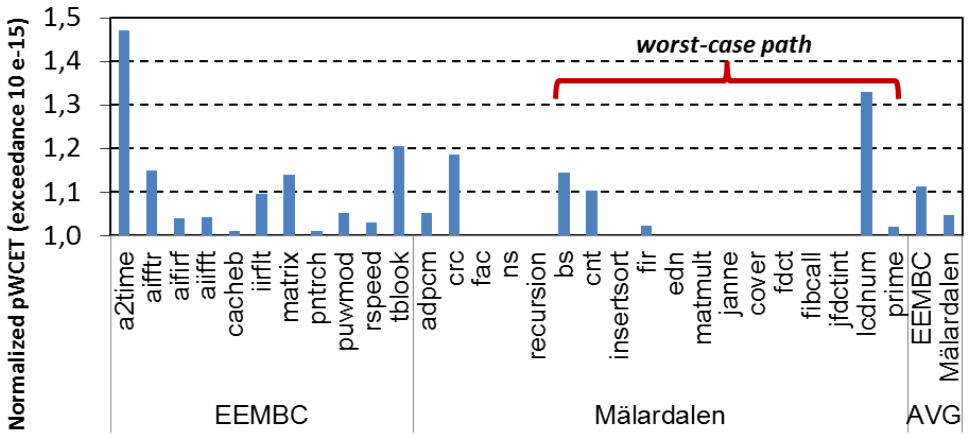


Figure 10.6: Impact on pWCET estimates of *PUB* with respect to MBPTA applied over the original program with the user-provided input vectors.

10.7.2 pWCET Estimates

In this section we compare the pWCET estimates obtained with *PUB* and MBPTA. Following the iterative method described in Chapter 3.1.2 we carried out 1,000 experiments.MBPTA then uses those execution time measurement and EVT to extract pWCET estimates. In all experiments we apply the Wald-Wolfowitz independence test and the two-sample Kolmogorov-Smirnov identical distribution test. We use a 5% significance level (a typical value for this type of tests). We also apply the ET test for Gumbel convergence testing. All three tests were passed for all EEMBC and Mälardarlen benchmarks.

PUBaa exploits the case in which there is little overlap between the (long) address sets in each branch of a CFC. This would lead to *PUBam* generating long extended versions of each branch with many accesses to all those addresses, which in fact generates worse PCSs than feasible with the original version of the code. Such a case has not been found in the benchmarks evaluated in this study, making *PUBaa* not to have any benefit over *PUBam*. As part of our future work, we aim at finding benchmarks that allow us to illustrate quantitatively scenarios where *PUBaa* outperforms *PUBam*.

Figure 10.6 shows the pWCET increase introduced by *PUB* with respect to the original MBPTA. We observe that for EEMBC and Mälardarlen the average pWCET slowdown is 11% and 5% respectively. Only three benchmarks suffer an slowdown between 20% and 50%. *a2time*, for instance, uses a data working set fitting quite well into the D-cache, so when applying *PUB* more data are accessed on each iteration, the working set exceeds cache size and execution time (and so pWCET) increases noticeably. However, most of the benchmarks experience low or even negligible pWCET degradation when applying *PUB*.

Similar to other timing analysis techniques, *PUB* benefits from information on infeasible paths. In particular, for *a2time*, *pntrch* and *ns*, we have used semantic knowledge of the infeasibility of some paths. We detected that some path combinations were infeasible by semantic construction of the program. This is the case of *a2time*, detailed in the next section, in which there is a sequence of `if-then` constructs out of which exactly one can be executed in any traversal. Therefore, since every execution is indeed triggering the worst (and only) timing behaviour, there is no need for balancing those `if-then` constructs. If this semantic information is not provided by the user, *PUB* assumes that all CFC can be executed in each traversal, thus producing some non-negligible and 'artificial' overhead. If the user prevents *PUB* from balancing CFC in which some path combinations are deemed to be infeasible the overheads of PUB can be reduced.

10.8 Exploiting User Knowledge to Reduce pWCET

In this section we discuss the exploitation of user knowledge about the application in order to reduce the *PUBam* overhead which is caused by infeasible paths.

Applications can have path combinations that are not possible to be exercised under any circumstances at deployment time. An example of such combinations are paths that come from different operation modes in the application, e.g., take-off and cruise mode. Another case appears when there are conditional paths in the program, whose conditions are impossible to be satisfied at the same time. In both cases, when applying timing analysis in an application agnostic way, if the program's Worst Case path includes paths from infeasible combinations, it will result in an overestimation of the WCET. To overcome this problem the WCET either needs to be computed in a per-operation mode manner, or by not taking into account those paths that cannot be taken together. As an example of this behaviour we present the `a2time` benchmark of the EEMBC automotive suite.

The structure of this program follows the pattern of EEMBC automotive benchmarks: a loop with a specified number of iterations. The loop body is unrolled explicitly 3 times, which makes the sequence of input sensing and calculations inside the loop body to be repeated 3 times.

In Figure 10.7 we can see the pseudocode of the `a2time`'s loop body. The compiler cannot know that the 8 paths are mutually exclusive, which is actually the case. Therefore *PUB* will balance all paths with an else construct.Since exactly one of the 8 conditions is true, this will cause the program to execute 7 expensive else paths unnecessarily (multiplied by 3 as the loop is unrolled 3 times), as those paths are infeasible. This results in a 2.3X times increase in the execution time.

Users knowledge about the application can be used to reduce this overhead. This can be done by: a) using compiler directives, in order to instruct the compiler

```
for i < ITERATIONS do
    if angleA ≤ angle < angleB then
        firing_time = complex math expression 1 // Cylinder 1
    end if
    if angleC ≤ angle < angleD then
        firing_time = complex math expression 2 // Cylinder 2
    end if
    ...
    if angleO ≤ angle < angleP then
        firing_time = complex math expression 8 // Cylinder 8
    end if
    // Same blocks repeated 2 more times
end for
```

Figure 10.7: a2time loop structure

```
for i < ITERATIONS do
    if angleA ≤ angle < angleB then
        firing_time = complex math expression 1 // Cylinder 1
    else if angleC ≤ angle < angleD then
        firing_time = complex math expression 2 // Cylinder 2
        ...
    else if angleO ≤ angle < angleP then
        firing_time = complex math expression 8 // Cylinder 8
    end if
    // Same blocks repeated 2 more times
end for
```

Figure 10.8: a2time loop structure with simple code restructuring to help the compiler identify mutual exclusive paths

not to balance these paths; or b) restructuring the code to make clear to the compiler that these paths are exclusive to each other.

The former requires using compiler directives (e.g., `# pragma no_balance`) and has the advantage that keeps the application unmodified, which is of particular benefit in case of legacy applications that are already certified.

The latter consists in a simple restructuring of the application in order to make explicit that the paths are infeasible. For example, in Figure 10.8, the individual **if** statements are changed to **else if** conditions, which shows that only one of the 8 paths can be true at any moment. This way, *PUBam* would only add a single balance path in this construct, which would be never exercised, so no overhead would be induced. Application of *PUBam* to the rest of the paths in the benchmark, leads to a 47% increase in the WCET, compared to the 2.3X without taking advantage of knowledge about the application.

10.9 Related Work

Several approaches have been proposed to deal with the complexity of WCET analysis when different input data cause the code to execute on different execution paths with differing execution times. The Single-Path approach [Puschner (2003)][Puschner (2005)] transforms CFC into a sequential set of instructions based on the concept of *predicated execution* [Mike & Schlansker (1991)]. Predicated instructions have a *predicate*, such that the instruction is only executed if its predicate evaluates to true, otherwise the processor assumes it to be a nop operation. For instance, for an *if-then-else*, the instructions in each branch are changed by sequential code with conditional-move assignments for each of the conditionally changed variables. *PUB* does not make the execution of a program single-path, but instead changes the branches in conditional CFCs such that extended paths take longer to execute than any of the original branches of the conditional CFC. Further note that the extended code is only used at analysis time; at deployment time the original code is used, hence adding no overhead on average performance.

Several studies such as [David & Puaut (2004)] focus on probabilistic schedulability analysis and assume, for each task to be scheduled, a known distribution of execution times. Other studies focus on WCET estimation such that WCET estimates are probabilistic, which is our focus. Some of those studies focus on time-deterministic architectures and assume that the external events affecting the execution time of a program, i.e. input data vectors, are *assumed* to be random variables. Based on this assumption authors associate to each potential execution path of the program a probability. To this end, each condition is associated a probability. For instance in the statement `if (a > b) then S1; else S2;` if a and b are input data, these approaches assume that it is known the probability of the condition to be true, and hence the probability S1 to be executed, can be computed. This puts high requirements on the user to provide input data that is probabilistically representative of deployment time behaviour. In that respect, it is worth mentioning that the primary goal of MBPTA is to provide pWCET estimates that hold under execution conditions that may occur during actual operation: whereas those conditions may not be exactly identical to those captured by the observation runs made at analysis time, analysis time conditions must still reproduce or upper-bound the probabilities of the execution times that may occur during operation [Cazorla *et al.* (2013b)]. MBPTA controls the hardware and the software behaviour, for instance by proposing time-randomised caches, so that the requirements on the user to provide representative execution conditions is heavily reduced [Cazorla *et al.* (2013b)].

Other authors [Lu *et al.* (2012)] let the application run long enough on a time-deterministic architecture collecting the observed execution times. Those execution times are then randomly sampled to apply Extreme Value Theory. However, the representativity of the obtained results (i.e. pWCET estimates) depends on how representative the collected observed execution times are, with respect to the application's execution times, and those techniques provide no guarantees on that matter [Cazorla *et al.* (2013b)].

Authors in [Liang & Mitra (2008)] define the concept of probabilistic cache behaviour on a deterministic fully-associative cache. To that end, for every point in the program the (static) deterministic cache state in which the program is at that point is associated the probability of reaching that program point. On the one hand, authors also assume that the events affecting the execution time are random so that a probability can be associated to each path with the casuistic shown above. On the other hand, this differs from our concept of probabilistic cache state, in which *each access* has a distinct associated probability of hit and execution paths have no associated probability.

In addition to previous related works that we have described above, there are also relevant works that build on top of our PUB contribution and improve it in order to reach a high TRL level. In particular, Extended Path Coverage or EPC [Ziccardi *et al.* (2015)] is a technique based on the same concept as PUB, to remove the burden of input vector from the end user in order to compute the WCET. Similarly, EPC works on time-randomised caches, however it does not require changes to the compiler, thus reducing its implementation cost. Instead of padding the application code, it modifies the probability distribution of each path to negatively compensate for any potential benefit from the non-exercised paths that may lead to pWCET underestimation. In the PROXIMA project, EPC has been implemented in a commercial measurement-based timing analysis tool [Rapita Systems (2008)] and has been assessed with an industrial case study [Mezzetti *et al.* (2017)].

10.10 Summary

Obtaining trustworthy and tight worst-case execution time (WCET) estimates is of prominent importance in safety-critical real-time embedded systems. However, the WCET estimation process has to be affordable for the user to apply it. Measurement-Based Probabilistic Timing Analysis (MBPTA) has emerged recently to respond to this challenge by providing means to easily determine accurate WCET estimates. Unfortunately, MBPTA still relies on the user providing path coverage, which is often beyond of what the user can provide.

In this chapter we propose a Path Upper-Bounding method, *PUB*, which works in conjunction with MBPTA. *PUB* allows deriving pWCET estimates for all paths of the program, including those not exercised by the input vectors given. This makes that the coverage needed for functional verification is also enough for the timing analysis, thus drastically reducing the cost of timing analysis. Our results show that *PUB* increases pWCET estimates only by 5% and 11% on average for Mälardarlen and EEMBC with respect to MBPTA.

Chapter 11

Probabilistic Timing Composability

11.1 Introduction

In this chapter we address an important issue related to the use of time-randomised caches in Integrated Architectures. Those systems, as discussed in the Introduction are increasingly used in safety critical systems, in order to accommodate more complex software functions of mixed criticality.

Due to the critical nature of these systems, placing multiple functions – possibly at different criticality levels – on the same hardware, requires *space isolation* and *time isolation* [APEX Working Group (2013)]. The former prevents any data-related misbehaviour of one function from affecting the data sources of other functions. The latter, which is our focus, ensures that the worst-case execution time (WCET) bound determined for one function is guaranteed (hence can never be exceeded) in the face of other functions competing for execution on the same processor. This interpretation of time isolation directly follows from the property known as *time composability* (TC). Achieving both forms of isolation enables a drastic reduction of development costs as each subsystem can be independently developed and then incrementally integrated and qualified in the system without risks of regression at system level [RTCA (2005)] [Wilson & Preyssler (2008)] [Elmqvist *et al.* (2008)] [International Organization for Standardization (2009)].

Classic timing and schedulability analysis assume time composability in the software components that constitute the system. Their assumption however is safeguarded by conservative and pessimistic assumptions, which fatally defeat the industrial goal stated above.

State-of-the-art static timing analysis techniques are intrinsically limited by the complexity of constructing a sufficiently accurate model of the hardware and of the software executing on top of that hardware. Any inaccuracy or lack of the required knowledge about the hardware timing or the software execution behaviour may have an inordinate impact on the tightness of the resulting WCET estimates. As the hardware becomes increasingly complex and software functions increase in number and size, building accurate models of the hardware and software to determine tight WCET bounds becomes prohibitive, if at all feasible [Mezzetti & Vardanega (2011b)] [Kirner & Puschner (2008)] [Puschner *et al.* (2009)].

MBPTA, which has been the focus of this thesis, has been proven to be a viable method to perform timing analysis of programs in CRTES. However there is still a lack of understanding of what *MBPTA-conformance* – as defined in Chapter 4 – offers in the way of time composability. The question is how MBPTA tight yet safe WCET bounds can be in comparison to those obtained by state-of-the-art techniques.

The contributions of this chapter are the following:

1. We identify how time composability can be achieved in probabilistic systems while keeping pWCET estimates tight. We focus on the effect of cache memories, as they are the typical example of a hardware acceleration feature known that severely challenges classic WCET analysis.

2. We describe the information required from software components to be able to time compose their pWCET estimates. In particular, we show that the *reuse distance*[1] of memory accesses is enough to fully characterise the time composability properties of a software component.

3. We illustrate how to compose pWCET estimates in a processor set-up with complex cache configurations and provide methods to generate pWCET estimates suitable for composition in MBPTA systems. We demonstrate that smart compositions allow using tighter pWCET estimates of the components.

Using the simulation environment introduced in Chapter 3.1.2, our results show that MBPTA makes pWCET estimates attractive to time composability. In particular, we observe up to 25% reduction in our pWCET bounds with respect to the WCET obtained by flushing the processor state prior to program execution, which is the standard industrial practice to attain time composability.

[1]Reuse distance stands for the number of memory accesses in-between two consecutive accesses to a particular address.

11.2 Incremental qualification

For many real-time embedded systems, the hardware and software components of the system have to be developed in parallel and integrated incrementally. A key design principle to contain verification costs in Integrated Architectures such as AUTOSAR for automotive [AUTOSAR (2006)] and Integrated Modular Avionics (IMA) for avionics [SAE (2001)] is to secure the possibility of *incremental qualification* [RTCA (2005)] [Wilson & Preyssler (2008)] [Elmqvist *et al.* (2008)] [International Organization for Standardization (2009)], whereby each software component can be subject to verification and validation – including timing analysis – in isolation. This goal is achieved by guaranteeing that no interaction occurs between isolated functions, in time and space, in their sharing of execution resources. At functional level, this translates into providing for space isolation, such that no misbehaving function may corrupt the data used by other functions. At timing level, this requires *time composability*, such that the timing behaviour of a function (in terms of its known lower and upper bounds) is not affected by the execution of other functions, whereby software functions enjoy time isolation. This property determines that the timing behaviour of an individual component does not change in the face of composition with other components.

11.3 Time Composability

Time composability applies to the interaction between the Operating System (OS) and the user-level application or between the user level application themselves.

Regarding the interaction between applications and the OS, in [Baldovin *et al.* (2012)] the authors present the design of a time composable OS. In the cited work, the latency of every system call that may affect the pWCET of application-level program units is designed to be constant and to not perturb the state retained in the processor at the time of the call: in this manner the execution of the OS has no effect on pWCET bounds of the caller program unit.

In this study we focus on time composability (TC) at application level. The TC we seek (i) allows tight pWCET estimates to be obtained for program units and (ii) incurs affordable design, implementation and verification costs. We dismiss approaches to TC that hinder functional design and scalability: for example, the use of static (table-driven) scheduling for the execution of program units and making pessimistic allowances for the execution slots assigned to them.

Next we describe the typical structure of industrial-quality user-level CRTES applications and then discuss how state retention in hardware, with focus on cache memories, and in software components threatens TC. Finally, we state the problem and the assumptions on top of which we build our case for probabilistic time composability.

Algorithm 1 Static schedule of main procedures with run-to-completion execution semantics (top) and Example control flow of a $Main\_procedure_i$

void $Main\_procedure_i()$ {
$Inner\_procedure_{i,1}()$
$Inner\_procedure_{i,2}()$
$Inner\_procedure_{i,1}()$
for $(j = 0;\ j \leq 5;\ j++)$ **do**
    $Inner\_procedure_{i,3}()$
    $Inner\_procedure_{i,4}()$
    $Inner\_procedure_{i,2}()$
end for
}

---

while true **do**
    $Main\_procedure_1()$
    $Main\_procedure_2()$
    ...
    $Main\_procedure_n()$
end while

11.3.1 Software Structure of Real-Time Functions

We regard the software design to include a collection of independent *main procedures* each of which in turn contains a number of *inner procedures*. Main procedures are called indefinitely as long as the system is operational according to some system level schedule (see Algorithm 1). We consider inner procedures to be the unit of software development and verification. Integration and validation occur incrementally.

The system-level schedule determines how the execution of main procedures interleave with one another. The Control-Flow Graph of each main procedure determines the way inner procedures are called and what interleaving occurs between any two successive executions of any of them. However, as inner procedures are the unit of parallel development hence also the Unit of Composition (UoC), their exact internal structure is only known late in the development process.

We term *Program Unit* the software component on which pWCET analysis is to be performed. This can be at the granularity of either the main procedure or the inner procedure. The pWCET value that matters for system-level scheduling is that of main procedures, which in turn results from the pWCET bounds determined for its inner procedures. It is therefore safe to assume that pWCET analysis operates on inner procedures and proceeds upward from them.

11.3.2 Problem Statement and Assumptions

The main factor affecting TC in the execution of a UoC is the state retained in the processor hardware (e.g., cache contents) and in the software (e.g., data on which path decisions are dependent). We ignore software state issues because MBPTA is capable of handling the impact of software state in the determination of pWCET estimates [Cucu-Grosjean *et al.* (2012)].

We concentrate on the processor state whose retention may affect TC. Given a UoC B representing an inner procedure to which MBPTA is to be applied to obtain a pWCET estimate, we assume that (i) a time-composable OS is used such that the latency of each OS service call is constant and its execution has no effect on the state retained in the target processor [Baldovin *et al.* (2012)]; and (ii) run-to-completion semantics is granted for all UoC so that the interference effects resulting from preemption do not need to be accounted for in the analysis. Assumption (ii) matches current IMA practice in civil avionics [Watkins & Walter (2007)].

Let us now consider two consecutive execution instances of B. Let us call them B_p and B_q respectively, with the subscript representing the ordinal execution number of the UoC instance, with $q > p$, so that B_p precedes B_q. The amount of useful processor state that B_q can reuse from B_p and how much benefit this reuse can cause on the execution time of B_q, and thus on its pWCET, depends on: (1) How much internal reuse B makes, which we call *intrinsic reuse* and which depends on the size of B's working set. (2) The execution 'duration' of B. The longer the execution the lower the relative effect that different initial conditions can have on B_q. (3) The size of the cache(s) that retains B_p's state in hardware. And (4) what portion of B's working set is not evicted by the code executed between B_p and B_q. We term that foreign code *disturbing code* and its effect *state disturbance*.

Standard analysis of the Control-Flow Graph of the main procedure to which B belongs allows determining when different inner procedures are called (see Algorithm 1).

An easy yet pessimistic way to attain time composability for B is to compute its pWCET assuming that all processor state is flushed prior to its every execution. This assumption has the advantage that it makes no assumption on the code executed before B is called. Yet, as we show later in Section 11.5, this approach to time composability may introduce significant degradation in both average and guaranteed performance (pWCET) for B. Achieving time composability in this manner allows too little load in the system, which goes counter the industrial need presented in Section 11.1.

We want to allow useful processor state retention to be considered in the determination of tight pWCET estimates and want to do so in a manner that achieves time composability and that is economically viable to implement. The partic-

ular problem we solve can be formulated as follows: *provide hardware/software mechanisms such that the pWCET estimate for a UoC stays valid in the face of composition with any other UoC during system integration. These hardware/software mechanisms must not require any change in existing MBPTA techniques, applied to MBPTA-conformant processor architectures.*

11.4 Probabilistic Time Composability

Previous studies [Cazorla *et al.* (2006)] show that when flushing the processor state, programs recover their core (pipeline) state – including the branch predictor state – in a few hundred cycles. This is not true for caches instead: depending on the cache size in fact, it may take an amount of time several orders of magnitude higher than core state recovery, before the working set is restored in the cache after a flush. For this reason in this work we focus on on-chip cache resources and instead simply flush the core (pipeline) on every entry to a UoC.

We assume time-randomised caches on which there is a probability for each cache access to hit or miss in cache, as needed by MBPTA, so its timing behaviour can be modelled with an ETP: $ETP(i) = \{t_{hit}, t_{miss}\}, \{p_{hit}, p_{miss}\}$, where t_{hit}, t_{miss} express the latency in case of a hit and an miss respectively, and p_{hit}, p_{miss} the associated probabilities. This can be achieved for both set-associative and fully-associative caches implementing random placement and replacement policies as we explained in Chapter 5.

On time randomised caches, on every access that incurs a miss, a cache line is randomly evicted from cache as the new location in which the fetched line is placed, i.e the cache set and the cache way, is randomly selected. We showed in Chapter 5 that given any two accesses to a given address $@_A$, the number of evictions occurring between those two accesses affects the hit probability of the second instance of $@_A$. In fact, the higher the number of misses, the lower the hit probability of the second occurrence of $@_A$. The reason is that the probability of selecting (and so evicting) the cache set and the cache way in which $@_A$ resides increases.

We note that the number of evictions is upper-bounded by the number of accesses occurring both accesses to $@_A$. However, upper-bounding the number of misses of the disturbing code with the number of cache accesses is overly pessimistic since only a subset of the accesses are misses. For the specific sake of time composability, we seek a set of metrics me that characterise a program unit such that given two disturbing codes dc_1 and dc_2, we can prove that the cache ageing caused by dc_1 is higher than that caused by dc_2 if dc_1 produce worse effects than dc_2 on the same program unit for all me. That is, for $1 \leq i \leq M$ where M is the number of metrics to consider of a program unit, if $me_i^1 \geq me_i^2$ the effect of dc_1 is

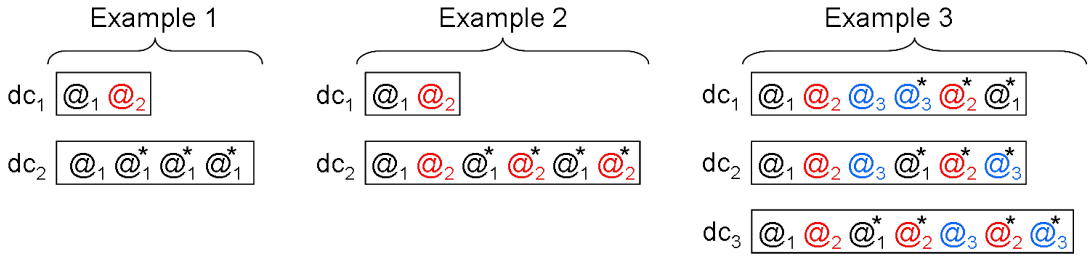


Figure 11.1: The effect of the number of accesses, the number of unique addresses and reuse distances. Accesses marked with * have non-infinite reuse distance.

worse than the effect of dc_2. Under this situation, if we obtain a pWCET bound for B_q under the composition scenario $B_p dc_1 B_q$ we know that such pWCET bound is a true upper-bound value for the $B_p dc_2 B_q$ composition scenario.

In Figure 11.1 we show several examples that illustrate which metrics (me) we use.

Example 1 A large number of accesses of a given disturbing code does not guarantee that it is an upper-bound for any other disturbing code with fewer accesses. In particular dc_1 performs only 2 accesses that produce 2 misses, whereas dc_2 performs 4 accesses that produce just 1 miss since the last 3 accesses are guaranteed to hit. The number of unique addresses[1] is the critical factor here since the first access to any given address will miss and produce an eviction, whereas other accesses may miss (marked * in the figure) or hit if the data they look for are still in cache.

Example 2 While the number of unique addresses matters, the number of accesses also does. In particular, dc_2 will produce at least as many evictions as dc_1. In fact, both dc_1 and dc_2 can evict up to 2 of the cache lines left in cache by B_p. We deepen on this issue later in this section.

Example 3 Reuse distance also matters. In particular, dc_1 and dc_2 access the same addresses the same number of times, but in a different order. However, reuse distances are different. Their respective reuse distances are $(\infty, \infty, \infty, 0, 2, 4)$ and $(\infty, \infty, \infty, 2, 2, 2)$ respectively[2], which have different impact on the hit probability of accesses. It is therefore unclear how to determine which disturbing code upper-bounds others. Similarly, it is unclear whether dc_3 bounds dc_1 and dc_2 even if it has a larger number of accesses and the same number of unique addresses.

[1] Unique addresses are those remaining once repetitions are removed.

[2] ∞ denotes the first access to a given memory location in the sequence,

Reuse distance. Given a sequence of accesses to cache $@_A@_B@_C@_D...@_A$, the hit probability of the second instance of $@_A$ depends on the *number of intermediate accesses occurring in between the first and the second occurrence of* $@_A$ *and the reuse distance of each of those accesses*: $rd_B, rd_C, rd_D, ...$ We define the reuse distance for an access $@_A$ as the number of memory accesses occurred since the last access to the same address. For instance, in dc_1 in Example 3 of Figure 11.1, the reuse distance is 4 for the second instance of $@_1$. The reuse distance of an access determines its hit probability: the higher the reuse distance, the lower the hit probability of that access. In the extreme case, when the reuse distance an access is infinite its hit probability is 0. We call such an access a *unique access*. The hit probability of the second instance of $@_A$ is inversely proportional to the miss probability of intermediate accesses, which in turn depends on the reuse distance of those accesses.

Determining the reuse distance of two subsequent executions of a UoC B allows obtaining the pWCET estimates of the second execution of B such that it benefits from the cache state left by the previous execution of it. These pWCET estimates must be obtained making the least assumptions on the disturbing code executed in between those instances of B.

Hence, our first approach considers the reuse distances of the instruction and data accesses of the disturbing code: $me = \{rd_i, rd_d\}$. Given two disturbing codes, dc_1 and dc_2, we say that the former produces worse interference than the latter if each instruction and data access of dc_2 can be paired up with an instruction and data access respectively of dc_1 with higher reuse distance. The rationale behind this approach is that the higher the reuse distance of an access, the higher its miss probability thus the higher the probability of an eviction, which reduces the survivability of cache contents. In other words, if dc_1 performs at least as many instruction and data accesses as dc_2 and the miss probabilities for dc_1 accesses are higher than those for dc_2, dc_1 upper-bounds dc_2 cache ageing. An easy mechanism to check how the reuse distances for data and instructions for dc_1 upper-bound their counterparts for dc_2 is shown in Algorithm 2.

Below we show some examples of reuse distance vectors for dc_1 and dc_2. In each case we show when dc_1 upper-bounds dc_2. For this example, we only consider data accesses, but the same considerations apply to instruction accesses.

- dc_1 upper-bounds dc_2: $rd_1 = \{7, 5, 3, 2\}$, $rd_2 = \{6, 5, 2\}$. $7 > 6$, $5 \geq 5$, $3 > 2$, $2 > \emptyset$. Hence, dc_1 upper-bounds dc_2.

- dc_1 does *not* upper-bound dc_2: $rd_1 = \{9, 8, 7, 0\}$, $rd_2 = \{1, 1, 1, 1\}$. $9 > 1$, $8 > 1$, $7 > 1$, $\emptyset < 1$. No reuse distance of dc_1 is paired up with the last one of dc_2, so dc_1 cannot be proven to upper-bound dc_2.

Algorithm 2 Checking whether disturbing code dc_1 upper-bounds another disturbing code dc_2 for MBPTA with EoM

```
Check if dc1 upper-bounds dc2
C = set of caches in the processor
for all c_i ∈ C do
    r1 = reuse distances for dc1 in c_i
    r2 = reuse distances for dc2 in c_i
    if |r2| > |r1| then return false
    end if
    r1sort = sort r1 from higher to lower
    r2sort = sort r2 from higher to lower
    for all r2sort_j ∈ r2sort do
        if r2sort_j > r1sort_j then return false
        end if
    end for
end for
return true
```

Although the approach described so far is conceptually applicable and can produce tight upper-bound values, modelling the reuse distances of all the disturbing code that may possibly execute between B_p and B_q is too complex. It would be interesting, therefore, to find alternative approaches to bound the effect that a given *dc* can cause on different instances of a UoC B, which can reduce the characterisation information that the user has to provide about the disturbing code. Next we present one such approach.

Using the number of unique accesses One way to achieve tight and safe bounding consists in having the user provide the number of unique accesses u_i, u_d that the *dc* to be bound may perform. Let's assume that dc_1 performs the following sequence of accesses $@_A@_B@_A@_B$, so the number of accesses (N) is 4 and the number of unique accesses (u) is 2. Next, we show that such a program can evict at most u cache lines of the contents stored prior to its execution. The first time $@_A$ is accessed it misses in cache evicting data of B_p, which we call *prior data*. The first access to $@_B$ will also miss but it can evict data of B_q or $@_A$. The second access to $@_A$ will only cause a miss if $@_B$ evicted A and no element of B_p is in cache. Analogously, the second access to $@_B$ will only evict data in cache from B_p only if the first instance of $@_B$ evicted the first instance of $@_A$ and the second instance of $@_A$ evicted the first of $@_B$. *Overall, a disturbing code of N accesses and u unique accesses can evict at most u elements that were in cache prior to its execution.* The reuse distance of the N accesses will determine the actual probability that u

prior data are evicted. For example, the sequence $@_A@_A@_A@_B@_B@_B$ that has the same N and u that the sequence $@_A@_B@_A@_B@_A@_B$ is much less likely to evict u prior data.

Given a dc_1 with u_i, u_d unique accesses, we want to synthetically generate a disturbing code dc_2 whose probability of evicting u_i and u_d prior data in the instruction and data cache respectively is higher than for any other dc_1 with u_i, u_d unique accesses regardless of the number of times they are accessed and the reuse distance of those accesses.

Note that the user is only asked to provide u_i, u_d for the disturbing code, which can be easily obtained by means of profiling at the integration stage.

Next, we must derive how a dc must look like so that it bounds a program whose number of unique instruction and data cache lines accessed is u_i, u_d. To that end, it can be proven that for a cache with S entries, the number of distinct entries evicted (dee) after l random evictions is as follows [Feller (1968)]:

$$dee = \left[1 - \left(1 - \frac{1}{S}\right)^l\right] \times S \tag{11.1}$$

In other words, if we want to evict at least u distinct entries so that $dee = u$, the number of evictions required can be obtained as follows:

$$l = \left\lceil \frac{\log\left(1 - \frac{u}{S}\right)}{\log\left(1 - \frac{1}{S}\right)} \right\rceil \tag{11.2}$$

Therefore, a dc causing at least l evictions bounds the impact of any program with up to u unique accesses.

11.4.1 Software Support

With MBPTA, we make observation runs on the target system that capture the effect of disturbing code on the UoC of interest. Hence, disturbing code cannot be solely a conceptual artefact but it has to be concretely created at either hardware or software level.

We need to implement simple programs, which we term *micro-benchmarks*. A micro-benchmark produces a sequence of accesses to each cache memory such each access addresses a different cache line. As obtaining that effect for data and instruction caches simultaneously is difficult, a first part of the micro-benchmark can produce evictions for data and a second part of it can cause evictions for instructions. The number of evictions required in each cache is determined with equations 11.1 and 11.2. Data evictions simply require a loop with a load instruction whose stride is equal or higher than the largest cache line in any data

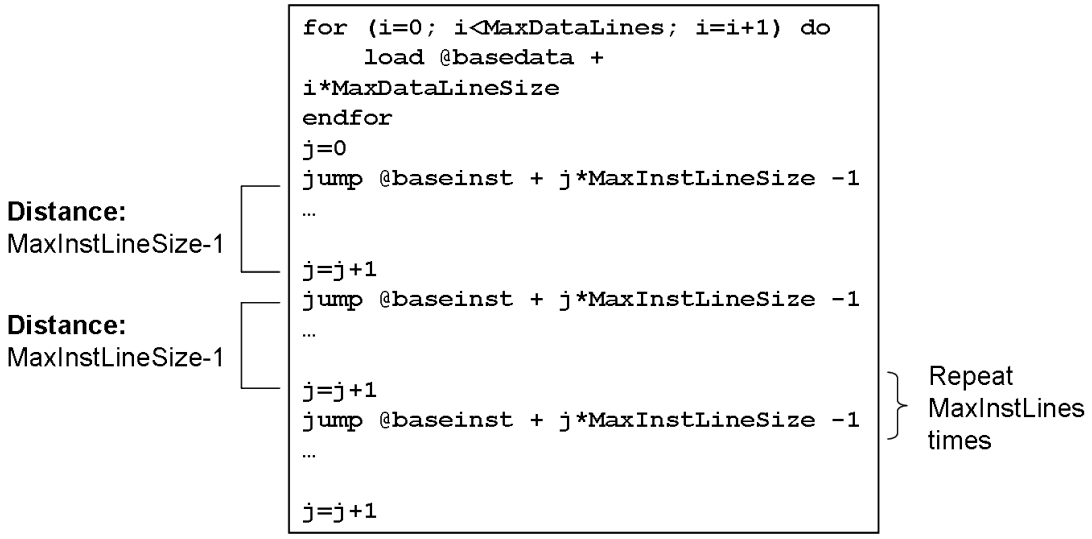


Figure 11.2: Example of a micro-benchmark

cache. For instance, if we use a data cache with 32-byte cache lines and a data TLB (translation look-aside buffer) with 1KB pages, performing max_d loads with a distance of at least 1KB ensures that no reuse occurs in any of the data caches across those max_d accesses. A similar effect can be produced for instructions by executing branches with a stride equal or higher than the largest cache line in any instruction cache.

An example of how to build such a micro-benchmark is shown in Figure 11.2. First, a loop performs all the needed data cache evictions. The micro-benchmark code iterates as many times as data cache evictions are required, max_d, ($MaxDataLines$ in the figure) accessing data with a stride matching the maximum data cache line size ($MaxDataLineSize$). Note that data space must be allocated to guarantee that the micro-benchmark does not make any access beyond the program bounds. The second section of the micro-benchmark consists of linear code that jumps as many times as instruction cache evictions are required, max_i, ($MaxInstLines$ in the figure) with a stride matching the maximum instruction cache line size ($MaxInstLineSize$).

In order to produce composable pWCET estimates of $Main\_procedure_i()$ in Algorithm 1 therefore, we must analyse each instance of $Inner\_procedure_{i,j}$ against its disturbing code. For instance, if we focus on the second instance of $Inner\_procedure_{i,2}$, its disturbing code is $Inner\_procedure_{i,3}$ and $Inner\_procedure_{i,4}$.

The pWCET estimate for the second instance of $Inner\_procedure_{i,2}$ is determined by MBPTA [Cucu-Grosjean *et al.* (2012)]) as follows. We run $Inner\_procedure_{i,2}$ alone, then a particular instance of the micro-benchmark and finally the $Inner\_procedure_{i,2}$ again measuring its execution time in that last run.

To simplify that process, several values are chosen for max_d and max_i for the micro-benchmark are used. The higher the number of combinations considered, the tighter the pWCET, but for more experiments to run.

With that data, MBPTA produces the pWCET estimate considering the interference effect of the assumed disturbing code. This process is repeated for all disturbing codes considered. At integration time the user must select the lowest pWCET for those scenarios where the micro-benchmark considered bounds the real disturbing code, such that $max_d \geq l_d$ and $max_i \geq l_i$, where l_i and l_d are obtained with equation 11.2.

This approach to timing composability has the key property of being *oblivious to the particular location in memory where the data and instructions of the disturbing code are* because probabilistically analysable caches such as random placement and replacement caches break the structural relation between the address and location of cache contents.

11.5 Experimental Results

This section evaluates our Time Composability approach. First we introduce the experimental framework and show how cache size and disturbing code characteristics impact the survivability of cache lines. Then, pWCET estimates are obtained for several relevant benchmarks under different scenarios.

11.5.1 Experimental Framework

We use the simulation environment we described in Chapter 3.1.2. Both instruction and data caches are 8-way set associative with 16-byte lines. The hit latency for all caches is 1 cycle. The miss latency is set to 100 cycles, including the time to access main memory. This value is arbitrary but it serves the purpose of producing a significant jitter without being unrealistic.

We used a sample of Mälardarlen benchmarks [Gustafsson *et al.* (2010)], which are commonly used in the real-time community to evaluate WCET analysis tools and methods. Of those, we used: bs (BS), crc (CRC), qsort (QSO) and select (SEL). BS and CRC illustrate extreme cases with very large reuse and almost no reuse across executions respectively. QSO and SEL correspond to intermediate cases with moderate reuse across executions.

pWCET estimations and i.i.d. validation follow the procedure described in Chapter 3.1.2. The minimum number of runs we needed was always below $1,000$, and all execution time traces passed the i.i.d. tests successfully.

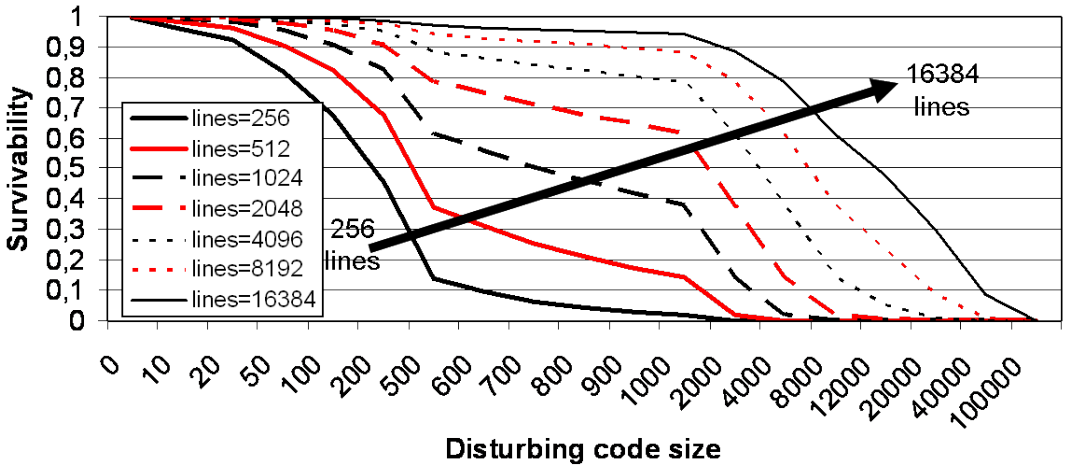


Figure 11.3: Survivability as a function of the number of unique accesses in the disturbing code for caches with different number of lines.

11.5.2 Results

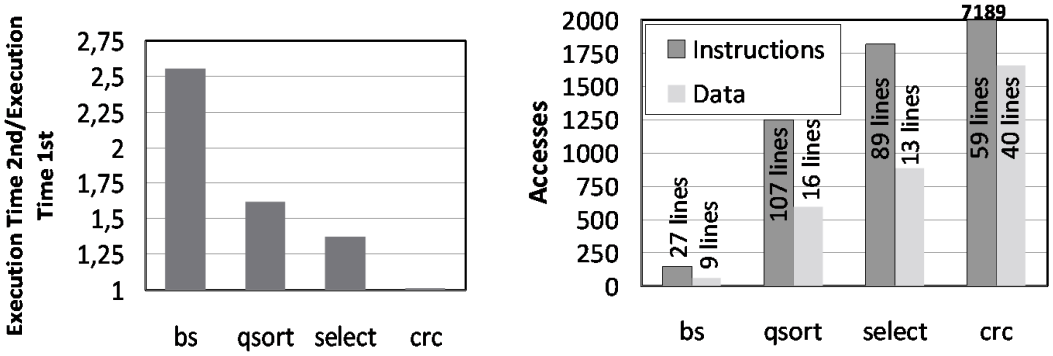
Survivability

We saw earlier that the potential reuse that B_q can make of the data and instruction state left by B_p is inversely proportional to the size of the disturbing code. To illustrate this point, we assume a cache in which after the execution of B_p all cache lines have reuse distance 0. Figure 11.3 shows the survivability of each line for a range of different cache sizes as we increase the number of unique addresses (and therefore also accesses) to the cache in the disturbing code. The cache size range corresponds to the typical number of cache lines for L1 and L2 caches.

Survivability with small-sized caches is low if all accesses of the disturbing code cause evictions. Conversely, for relatively large caches (e.g., in the order of some thousands of cache lines, which match the size of L2 caches) survivability increases noticeably and data in cache can be reused across execution instances. This is true even if the disturbing code accesses thousands of different cache lines, which is very unlikely for our target scenarios where small inner procedures are executed several times within their enclosing procedure (cf. Section 11.3).

Characterising the maximum benefit of TC

The potential benefit we can get with our approach to time composability depends, in addition to the intrinsic reuse of the application as shown in previous section, on the number of instruction and data accesses the functions in the disturbing code have.



(a) pWCET speedup (b) Instruction and Data access count

Figure 11.4: Characterisation of the Mälardarlen benchmarks used

Figure 11.4 shows the instruction and data accesses of the selected Mälardarlen benchmarks as well as the execution time reduction due to data and instruction reuse. The number of different cache lines accessed by each benchmark are shown on top of the corresponding columns for data and instructions. The experiments show that BS achieves the highest pWCET reductions ($\frac{pWCET\ empty\ cache}{pWCET\ zero\ disturbing\ code}$) since the number of accesses per cache line is low, hence the relative benefit of finding those lines in cache is significant. QSO and SEL show lower yet significant pWCET reductions because the relative impact of the lines reused across executions is lower. CRC shows the smallest reductions due to the relative low impact of the 99 (59+40) potential extra hits in a program performing almost 9,000 (7,189+1,655) total cache accesses.

TC of pWCET estimates

As mentioned in Section 11.4, flushing the processor state prior to the analysis of the UoC is the easiest way to TC. This solution prevents the execution time of the UoC under analysis from being affected from previous history of execution, which makes its WCET bound time composable, but at the cost of unnecessary pessimism. We refer to this approach as (*flush*,*flush*), meaning that we flush the instruction and data caches prior to execution.

Figure 11.4(b) shows that instruction access counts vary from a few hundreds to around 7,000 while the data access counts range between a few dozens to almost 1,700. Based on these values we use 6 different sets of values for u_i and u_d to build the micro-benchmark as described in Section 11.4: (u_i, u_d) = (*100*,*100*), (*200*,*200*), (*500*,*500*), (*1000*,*1000*), (*2000*,*2000*) and (*7000*,*7000*).

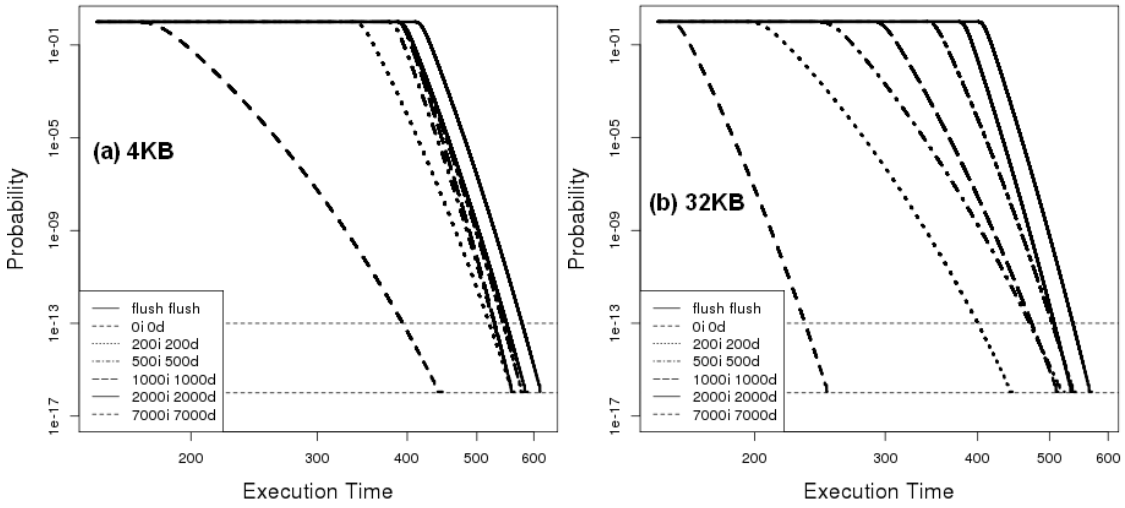


Figure 11.5: pWCET estimates obtained with MBPTA for different (u_i, u_d) values for the *bs* benchmark

	4KB cache				32KB cache			
evicts	bs	select	qs	crc	bs	select	qs	crc
(*0, 0*)	31.8	35.5	28.3	4.3	56.5	33.4	27.0	6.4
(*200,200*)	10.0	10.7	2.4	0.4	25.6	20.0	17.9	4.4
(*500,500*)	8.6	4.2	2.2	0.4	16.0	8.5	11.6	1.5
(*1,000,1,000*)	8.4	3.8	0.9	0.3	12.0	5.4	9.0	0.6
(*2,000,2,000*)	5.9	2.4	0.6	0.1	6.3	2.3	1.9	0.4
(*7,000,7,000*)	4.8	1.9	0.1$	0.0	5.6	0.5	0.9	0.4

Figure 11.6: pWCET percentage improvement (reduction) of (u_i,u_d) against (*flush*,*flush*) for 10^{-13} cutoff probability

Figure 11.5 shows the pWCET distribution for BS for various micro-benchmark settings. pWCET improvements diminish as the number of evictions performed increases, especially for small caches. Larger caches (e.g., 32KB) still provide significant pWCET improvements over the empty cache case despite the large number of evictions. Figure 11.6 details the results for all benchmarks for an exceedance probability of 10^{-13} per run. pWCET reductions are significant for some benchmarks as long as the number of evictions per cache does not exceed 1,000 evictions. Beyond that point benefits quickly diminish. However, given the context where probabilistic time composability is exploited (see Section 11.3), the disturbing code consists of small inner procedures.

Our representative benchmarks show that the number of unique cache lines accessed is 70 instruction lines and 20 data lines on average (see Figure 11.4 (b)).

cutoff probability	($flush$,0) icache flushing	(0,$flush$) dc flushing
10^{-13}	5.9%	22.8%
10^{-16}	6.4%	19.9%

Figure 11.7: Effect of instruction data and caches flushing (4KB cache)

By using the method in Equation 11.2 and a 4KB cache, the number of evictions required in instruction and data caches to safely upper-bound the effects of disturbing code would be 82 and 21 for instruction and data caches if only one inner procedure with 70 and 20 unique instruction and data lines accessed respectively is executed in between two consecutive executions of the UoC under analysis. If the number of such inner procedures increases, so does the number of evictions required. For instance, two such inner procedures would require 203 and 44 instruction and data unique accesses to bound their effect. As the number of inner procedures grows, their effect also grows, thus decreasing the pWCET improvement due to time composability. For instance, after 5 inner procedures we would not expect any cache line to be still in the instruction cache based on the method in equation 11.2 so we should simply assume that the instruction cache has been flushed. Conversely, the data cache would still provide some pWCET improvement as only 127 evictions would be needed.

The results are much better for the 32KB cache as it mitigates the impact of larger disturbing pieces of code. For instance, if only one inner procedure is executed in between two consecutive executions of the UoC under analysis, the number of instruction and data evictions required would be 72 and 21 respectively. Five inner procedures would only require 384 and 103 evictions respectively.

In summary, if the number of inner procedures between consecutive executions of the UoC under analysis is below 10, then the pWCET improvements will be in the range dictated by the (*200*,*200*) or (*500*,*500*) cases. Thus, significant pWCET improvements of 5%-10% can be observed for some inner procedures if caches are small (e.g., 4KB) and 10%-25% if caches are larger (e.g., 32KB) as shown in Figure 11.6.

Breaking down TC benefits across data and instruction caches

Flushing instruction and data caches has different effects on pWCET estimates. Figure 11.7 shows the average relative pWCET reduction across all benchmarks when only one cache is flushed, with respect to the pWCET estimate when both caches are flushed. As we can observe, when the instruction cache is empty, the pWCET reduction is quite small, in the range of 6%. On the other hand, if instruction cache contents are preserved, even in the complete absence of data in the data cache, the pWCET reduction is around 20%.

Thus, the maximum benefit is provided by the reuse of instructions, which is explained by the structure of the Mälardarlen benchmarks and is the typical behaviour of inner procedures in our context. In particular, their code consist of linear code with few small loops. Thus, the number of instruction cache lines fetched is relatively large (quite linear code) and cold misses account for most of the misses. This opens the door to a significant reuse across executions. Data sets are relatively small and highly reused inside inner procedures. Thus, there are fewer data cold misses, which decreases the relative impact of data reuse across executions.

11.6 Summary

MBPTA has been proven to enable the use of complex hardware acceleration features such as caches while providing tight guaranteed execution time bounds (WCET). The Time Composability properties of a MBPTA-conformant system however were not yet understood. As Time Composability is needed by embedded systems industry to enable incremental development, failing to provide arguments about how software components can be time composed defies the benefits of MBPTA.

In this chapter we have shown how program units executed on a MBPTA-conformant processor can be time composed. We have focused on the challenges to time composability caused by caches. In particular, we have considered time-randomised cache memories . We have shown that the amount of information required to characterise the disturbing effect of foreign code execution, which is needed to make the program unit time composable, is relatively low: the number of unique data and instruction accesses of the disturbing code.

This is in contrast with approaches based on deterministic architectures that require knowledge of all addresses for any potential disturbing code to determine which cache contents may be reused across executions.

Chapter 12

Conclusions and Future work

12.1 Contributions

Critical real-time systems have a constant need for higher guaranteed performance in order to stay competitive in the market and provide increased functional value. This need for increased performance can only be satisfied with a constant increase in hardware and software complexity, delivered in the form of more advanced processor designs than the ones traditionally used in the CRTES domain and with a dramatic growth in the amount of their code. At the same time, cost, power and weight constraints lead to the increased use of Integrated Architectures, which are based on the consolidation of many functionalities on a single platform. Timing Analysis is of paramount importance in these systems, comparable to their functional correctness. However, Deterministic Timing Analysis methods face scalability problems in the presence of this increased complexity.

In this Thesis we contribute in the state of the art of current and future CRTES towards the adoption of more complex hardware and software, by facilitating their analysis. We achieve this goal by focusing on the Measurement-Based Probabilistic Timing Analysis (MBPTA), which has been shown to have several advantages.

However, the flexibility of MBPTA comes with a price, since certain requirements need to be satisfied on the target platform, which is not the case with existing systems. This Thesis presents several hardware and software mechanisms that enable the use of MBPTA in current and future complex architectures – featuring advanced memory hierarchies such as the ones found in high-performance processors – in order to facilitate the wide adoption of this analysis method in the CRTES domain.

The contributions of this Thesis are divided in 3 main categories: a) hardware proposals for MBPTA, b) software solutions for MBPTA and c) timing analysis aspects of MBPTA on systems with caches.

In the hardware solutions, we target the design of future CRTES processors with 3 main contributions:

- The first contribution of this Thesis sets the foundations of the MBPTA-compliant processor design. We present a taxonomy of hardware resources based on their execution time *jitter* and which modifications are needed to their design to achieve MBPTA-conformance. In fact we show that these changes are few and can be easily achieved in current designs, either by enforcing worst case performance at analysis time or through randomising their timing behaviour.

- For the rest of the Thesis, we focus on the hardware resource with the highest contribution in average and worst-case performance as acknowledged by the real-time community, the cache. Based on the time-randomisation principle identified in the previous contribution, we propose efficient time-randomised set-associative cache designs which can be implemented in real hardware designs. In concrete, in addition to the random replacement policy, we propose a *random placement* policy, which is implemented by a means of a *parametric hash* function, able to provide different mappings across program executions, and therefore satisfying MBPTA's needs.

- Finally, by composing together single-level time-randomised caches, we extend the previous solution to arbitrarily complex memory hierarchies. We show that MBPTA can effectively analyse those complex designs, regardless their number of levels, inclusivity and write policy, including unified last-level caches, the analysis of which has been made possible for the first time in the literature.

Our software solutions focus on existing legacy systems, to enable the fast industrial adoption of MBPTA.

- Our first contribution in this direction is the proposal of a dynamic software randomisation (DSR) scheme, which provides the same properties of time-randomised caches on top of conventional caches. This is achieved by a combination of a compiler-pass and a runtime system, which provide random placement of program objects such as code, stack and global variables for each program run. Our proposal has been demonstrated to be scalable even with industrial size software, and despite its execution time overhead compared to the pure hardware solution it provides pWCETs competitive with it.

- In order to increase the industrial fitness of software randomisation, in our next contribution we address the issue of software certification which is essential in critical systems. Software using dynamic software randomisation (DSR) uses pointers and self-modifying code, which can complicate certification in certain CRTES domains. Therefore, we present a static method without a runtime, which is implemented in the linker and compiler level, offering the same advantages of DSR with respect to MBPTA properties, but more amenable to certification and with lower overheads.
- In our last software contribution, the static variant is refined further in order to work on source-code level, providing a solution against the tool qualification need that is imperative in CRTES. This way software randomisation does not require any change from industrial compilation toolchains and becomes completely independent from them, therefore facilitating the transfer of this technology to industry.

In the last part of this Thesis we address Timing Analysis aspects related to the presence of caches in MBPTA and its application in Integrated CRTES Systems.

- One of the most difficult processes in measurement-based techniques is the identification of worst case paths and inputs that exercise them. In order to simplify this process we propose PUB, a Path Upper Bounding technique working on time-randomised caches, which removes the requirement of the end-user to identify worst-case paths and inputs. This way PUB allows the computation of a WCET with MBPTA using a single input, that upper-bounds the pWCET of any program path.
- Finally, in the last contribution of this Thesis, we exploit a property of time-randomised caches in order to provide tighter WCET estimates in Integrated Systems. We propose *probabilistic timing composability* which allows software units to be time analysed in isolation, while retaining the validity of their pWCET estimations when composed together for the final system deployment. Furthermore, the probabilistic nature of time-randomised caches, enable system integrators to account for reuse between invocations of same software units, as opposed to current analysis methods on conventional hardware which require pessimistic assumptions such as empty cache.

12.2 Impact

With the previously mentioned contributions we cover extensively the area of complex architectures in CRTES and especially memory hierarchies in MBPTA, providing solutions close to industry needs. In particular, the initial contributions and

promising results of this Thesis obtained during the PROARTIS FP7 project, have served as a basis and helped to secure further funding for new projects and define a new research line within the CAOS group, towards MBPTA technology. Moreover, the increased scientific production of research articles within PROARTIS based on the contributions of this Thesis has helped to establish an increased interest of the real-time community in Probabilistic Timing Analysis and a wide acceptance of this method, having a significant impact on global level. As a consequence, recent editions of the top real-time conferences Real-Time Systems Symposium (RTSS) and Euromicro Conference on Real-Time Systems (ECRTS) among others, included special sessions in probabilistically analysable real-time systems.

As already mentioned in the external results section of each chapter, the proposed techniques have also been implemented in industrial tools and real hardware, and assessed against real avionics, automotive, railway and space case studies as part of several research projects funded by the European Commission (PROARTIS, PROXIMA) and the European Space Agency (Proartis4Space). This process not only allowed to corroborate the results of this Thesis on realistic setups, but also resulted in the production of a complete toolset that can enable MBPTA in practice and helped to increase the TRL of the developed tools to transfer this technology to industry.

Our time-randomised cache design has resulted in a patent owned by BSC, while our hardware proposals (MBPTA-compatible processor design and time-randomised caches) have been implemented in RTL-level in Cobham Gaisler's LEON3 processor. The design produced by both BSC and Cobham Gaisler, code-named LEOPARD, is currently available for licensing from Cobham Gaisler as part of their IP portfolio [Gaisler (2016)]. This processor, which is the first MBPTA-compatible hardware design in the market, has been also awarded with a HiPEAC technology transfer award in 2016.

Our dynamic software randomisation solution has reached the highest TRL compared to the rest of our software solutions. It has been ported to several architectures (PowerPC, SPARCv8), integrated with industrial RTOSes (SYSGO's PikeOS native, PikeOS ARINC 653, PikeOS RTEMS and RTEMS SMP) and tested on real boards (LEON3, LEON4, P4080) with several real industrial case studies (2 avionics, 2 aerospace, 1 automotive and 1 railway).

The static randomisation variant at source-level (TASA) has been validated with an automotive case study and ERIKA RTOS on the AURIX TriCore automotive microcontroller and on LEON3 using a variety of compilation toolchains (LLVM, gcc, HighTec). Currently, TASA works only with ANSI C, therefore it has lower TRL than dynamic software randomisation.

Both dynamic and static software randomisation solutions have attracted significant industrial interest, because they offer a faster exploitation path for MBPTA adoption. This interest has translated into on-going and future collaborations in new proposals for research projects, in order to increase further the TRL of those solutions, support new architectures and provide support for further aspects such as power and security, not only timing.

Other work in progress is performed in the context of other PhD and Master theses, some of which have led to joint publications, referenced in the Introduction. These include, but are not limited to, time-randomised multi-core architectures, with focus on shared resources among other cores such as shared caches, buses, networks-on-chip and memory controllers, as well as studies on the fundamental theory of MBPTA.

12.3 Future Work

Future work directions are endless, since this is a completely new field with a lot of unexplored possibilities. For example our PUB proposal opens the door to the research on easier-to-implement path coverage techniques over time-randomised caches. Researchers at the University of Padua and BSC, in collaboration with the Rapita timing analysis company are working on a technique called EPC (Extended Path Coverage) which achieves tighter pWCET than PUB, at the expense of block coverage instead of the full path coverage required by existing methods. EPC builds upon the principles identified in PUB and looks for achieving them without the need to modify the compiler.

Moreover, the benefits of the hardware and software designs devised on this Thesis have ramifications towards other problems such as security, power efficiency, etc., since randomisation is expected to provide also advantages on those fronts, and therefore research on these topics is expected in the future.

Finally, while the hardware solutions proposed in this Thesis are effective, further research on random placement and replacement implementations can be carried out. Moreover other hardware components can be studied in order to become MBPTA-compliant such as branch predictors, prefetchers, etc. Our vision in the long term is to reduce the performance gap of processors used in the CRTES domain and the ones used in everyday systems, so that they can both include high-performance features and be analysed using MBPTA.

References

Abella, J., Cazorla, F.J., Quiñones, E. & Vardanega, T. (2013). Measurement-Based Probabilistic Timing Analysis and i.i.d property. White Paper. http://www.proartis-project.eu/publications/MBPTA-white-paper. 41, 45, 53

Abella, J., Hardy, D., Puaut, I., Quiñones, E. & Cazorla, F.J. (2014a). On the Comparison of Deterministic and Probabilistic WCET Estimation Techniques. In *Euromicro Conference on Real-Time System (ECRTS-14)*. 7, 27, 29, 36

Abella, J., Quiñones, E., Wartel, F., Vardanega, T. & Cazorla, F.J. (2014b). Heart of Gold: Making the Improbable Happen to Extend Coverage in Probabilistic Timing Analysis. In *Euromicro Conference on Real-Time System (ECRTS-14)*. 27

Abella, J., Hernàndez, C., Quiñones, E., Cazorla, F.J., Conmy, P.R., Azkarate-askasua, M., Perez, J., Mezzetti, E. & Vardanega, T. (2015). WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *10th IEEE International Symposium on Industrial Embedded Systems, SIES 2015, Siegen, Germany, June 8-10, 2015*, 39–48. 4, 16, 18, 19, 20, 24, 26

Abella, J., Padilla, M., Castillo, J. & Cazorla, F.J. (2017). Measurement-Based Worst-Case Execution Time Estimation Using the Coefficient of Variation. In *ACM Transactions on Design Automation of Electronic Systems (TODAES)*. 25

Abeni, L. & Buttazzo, G. (1998). Integrating Multimedia Applications in Hard Real-Time Systems. In *Proceedings 19th IEEE Real-Time Systems Symposium*, 4–13. 23, 24

Agirre, I., Azkarate-askasua, M., Hernandez, C., Abella, J., Perez, J., Vardanega, T. & Cazorla, F.J. (2015). IEC-61508 SIL 3 Compliant Pseudo-Random Number Generators for Probabilistic Timing Analysis. In *2015*

Euromicro Conference on Digital System Design, DSD 2015, Madeira, Portugal, August 26-28, 2015, 677–684. 54, 66

Agirre, I., Azkarate-askasua, M., Saenz, I., Cros, F., Jegu, V., Maxim, C., (UoY), B.L., Kosmidis, L., Mezzetti, E. & Griffin, D. (2016). D4.8 Final Case Studies Experiments Results. *PROXIMA project deliverable*. 25, 99, 113

Altmeyer, S. & Davis, R.I. (2014). On the Correctness, Optimality and Precision of Static Probabilistic Timing Analysis. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, 26:1–26:6, European Design and Automation Association, 3001 Leuven, Belgium, Belgium. 26

Alvarez, L., Moreto, M., Casas, M., Castillo, E., Martorell, X., Labarta, J., Ayguade, E. & Valero, M. (2015a). Runtime-Guided Management of Scratchpad Memories in Multicore Architectures. In *2015 International Conference on Parallel Architecture and Compilation, PACT 2015, San Francisco, CA, USA, October 18-21, 2015*, 379–391. 31

Alvarez, L., Vilanova, L., Moreto, M., Casas, M., Gonzalez, M., Martorell, X., Navarro, N., Ayguade, E. & Valero, M. (2015b). Coherence Protocol for Transparent Management of Scratchpad Memories in Shared Memory Manycore Architectures. In *Proceedings of the 42nd Annual International Symposium on Computer Architecture, Portland, OR, USA, June 13-17, 2015*, 720–732. 31

APEX Working Group (2013). Draft 3 of Supplement 1 to ARINC Specification 653: Avionics Application Software Standard Interface. 174

ARM (2006). *Cortex-R4 and Cortex-R4F Technical Reference Manual*. 57, 66, 79, 107

ARM Ltd. (2013). Cortex-A series. `http://www.arm.com/products/processors/cortex-a/index.php`. 30

ARTEMIS, ITEA, and EUREKA (2015). Smart Industry: Impact of Software Innovation. Co-summit. 1

Atlas, A. & Bestavros, A. (1998). Statistical Rate Monotonic Scheduling. In *Proceedings 19th IEEE Real-Time Systems Symposium*, 123–132. 24

AUTOSAR (2006). *Technical Overview V2.0.1*. AUTomotive Open System ARchitecture. 127, 176

Axer, P., Ernst, R., Falk, H., Girault, A., Grund, D., Guan, N., Jonsson, B., Marwedel, P., Reineke, J., Rochange, C., Sebastian, M., Hanxleden, R.V., Wilhelm, R. & Yi, W. (2014). Building Timing Predictable Embedded Systems. *ACM Trans. Embed. Comput. Syst.*, **13**, 82:1–82:37. 20

Bailey, B. (2007). The Functional Verification of Electronic Systems. In *Publisher: International engineering consortium*. 127

Baldovin, A., Mezzetti, E. & Vardanega, T. (2012). A Time-Composable Operating System. In *12th International Workshop on Worst-Case Execution Time Analysis, WCET 2012, July 10, 2012, Pisa, Italy*, 69–80. 176, 178

Banakar, R., Steinke, S., Lee, B.S., Balakrishnan, M. & Marwedel, P. (2002). Scratchpad Memory: Design Alternative for Cache On-chip Memory in Embedded Systems. In *Proceedings of the Tenth International Symposium on Hardware/Software Codesign*, CODES '02, 73–78, ACM, New York, NY, USA. 31

Benedicte, P., Kosmidis, L., Quinones, E., Abella, J. & Cazorla, F.J. (2016). Modelling the Confidence of Timing Analysis for Time Randomised Caches. In *2016 11th IEEE Symposium on Industrial Embedded Systems (SIES)*, 1–8. 27

Berezovskyi, K., Santinelli, L., Bletsas, K. & Tovar, E. (2014). WCET Measurement-based and Extreme Value Theory Characterisation of CUDA Kernels. In *22nd International Conference on Real-Time Networks and Systems, RTNS '14, Versaille, France, October 8-10, 2014*, 279. 27

Berezovskyi, K., Guet, F., Santinelli, L., Bletsas, K. & Tovar, E. (2016). Measurement-Based Probabilistic Timing Analysis for Graphics Processor Units. In *Architecture of Computing Systems - ARCS 2016 - 29th International Conference, Nuremberg, Germany, April 4-7, 2016, Proceedings*, 223–236. 27

Berger, E.D. & Zorn, B.G. (2006). DieHard: Probabilistic Memory Safety for Unsafe Languages. In *Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, 158–168, ACM Press. 107, 108, 114, 117, 126, 145

Bernat, G., Colin, A. & Petters, S. (2002). WCET Analysis of Probabilistic Hard Real-Time Systems. In *IEEE Real-Time Systems Symposium (RTSS 2002)*, Austin, Texas (USA). 24

Bernat, G., Colin, A., Esteves, J., Garcis, G., Moreno, C., Holsti, N., Vardanega, T. & Hernek, M. (2007). Considerations on the LEON Cache Effects on the Timing Analysis of On-Board Applications. In *Proceedings of the Data Systems in Aerospace Conference (DASIA)*, vol. 638 of *ESA Special Publication*, 18. 31

Bhatkar, E., Duvarney, D.C. & Sekar, R. (2003). Address Obfuscation: An Efficient Approach to Combat a Broad Range of Memory Error Exploits. In *Proceedings of the 12th USENIX Security Symposium*, 105–120. 145

Bhatkar, S., Sekar, R. & DuVarney, D.C. (2005). Efficient Techniques for Comprehensive Protection from Memory Error Exploits. In *Proceedings of the 14th Conference on USENIX Security Symposium - Volume 14*, SSYM'05, 17–17, USENIX Association, Berkeley, CA, USA. 114

Boslaugh, S. & Watters, D.P.A. (2008). *Statistics in a Nutshell*. O'Reilly & Associates, Inc., Sebastopol, CA, USA, 1st edn. 71, 110

Bradley, J. (1968). *Distribution-Free Statistical Tests*. Prentice-Hall. 35, 50, 71, 110

Bünte, S., Zolda, M., Tautschnig, M. & Kirner, R. (2011). Improving the Confidence in Measurement-Based Timing Analysis. In *14th IEEE International Symposium on Object/Component/Service-Oriented Real-Time Distributed Computing, ISORC 2011, Newport Beach, California, USA, 28-31 March 2011*, 144–151. 22, 148

Butz, H. (2007). The Airbus Approach to Open Integrated Modular Avionics (IMA): Technology, Methods, Processes and Future Road Map. In *Workshop on Aircraft System Technologies*. 3

Cargal, J. (1988). *Discrete Mathematics for Neophytes: Number Theory, Probability, Algorithms, and Other Stuff*. 60

Cazorla, F.J., Knijnenburg, P.M., Sakellariou, R., Fernandez, E., Ramirez, A. & Valero, M. (2006). Predictable Performance in SMT processors: Synergy Between the OS and SMTs. *IEEE Transaction on Computers*, **55**. 179

Cazorla, F.J., Quiñones, E., Vardanega, T., Cucu, L., Triquet, B., Bernat, G., Berger, E.D., Abella, J., Wartel, F., Houston, M., Santinelli, L., Kosmidis, L., Lo, C. & Maxim, D. (2013a). PROARTIS: Probabilistically Analyzable Real-Time Systems. *ACM Trans. Embedded Comput. Syst.*, **12**, 94. 5, 24, 25, 26, 27, 28, 31, 39, 40, 56, 79, 92, 93

Cazorla, F.J., Vardanega, T., Quiñones, E. & Abella, J. (2013b). Upper-bounding Program Execution Time with Extreme Value Theory. *International Workshop On Worst-Case Execution Time Analysis (WCET 2013)*. 25, 27, 29, 43, 47, 171, 172

Charette, R.N. (2009). This Car Runs on Code. In *IEEE Spectrum online*. 3

Cobham Gaisler (2005). *Leon3 Processor*. Cobham Gaisler, http://gaisler.com/index.php/products/processors/leon3. 136

Cobham Gaisler (2011). *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*. 30, 57, 66, 79, 95, 107, 163

CONCERTO (2016). *ARTEMIS JU*. http://www.concerto-project.org/. 144

Cousot, P. & Cousot, R. (2004). Basic Concepts of Abstract Interpretation. In *Building the Information Society, IFIP 18th World Computer Congress, Topical Sessions, 22-27 August 2004, Toulouse, France*, 359–366. 7, 19, 20

Croxford, M. & Sutton, J. (1996). Breaking Through the V and V Bottleneck. In *Proceedings of the Second International Eurospace - Ada-Europe Symposium on Ada in Europe*, 344–354, Springer-Verlag, London, UK, UK. 4

Cucu-Grosjean, L., Santinelli, L., Houston, M., Lo, C., Vardanega, T., Kosmidis, L., Abella, J., Mezzetti, E., Quinones, E. & Cazorla, F. (2012). Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *24th Euromicro Conference on Real-Time Systems (ECRTS) 2012*, 91–101. 5, 10, 24, 25, 27, 28, 29, 31, 35, 39, 40, 43, 50, 51, 56, 59, 71, 74, 75, 79, 81, 91, 96, 97, 105, 110, 122, 178, 184

Curtsinger, C. & Berger, E.D. (2013). STABILIZER: Statistically Sound Performance Evaluation. In *Proceedings of the Eighteenth International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS '13, 219–228, ACM, New York, NY, USA. 107, 108, 114, 144

Dasari, D., Nelis, V. & Andersson, B. (2011). WCET Analysis Considering Contention on Memory Bus in COTS-based Multicores. In *ETFA2011*, 1–4. 17

David, L. & Puaut, I. (2004). Static Determination of Probabilistic Execution Times. In *Proceedings of the 16th Euromicro Conference on Real-Time Systems*, ECRTS '04, 223–230, IEEE Computer Society, Washington, DC, USA. 79, 171

Davis, R.I., Santinelli, L., Altmeyer, S., Maiza, C. & Cucu-Grosjean, L. (2013). Analysis of Probabilistic Cache Related Pre-emption Delays. In *Euromicro Conference on Real-Time System (ECRTS-13)*. 26, 47

DeGroot, M. & Schervish, M. (2002). *Probability and Statistics*. Addison-Wesley, Reading MA. 36, 50

Deverge, J. & Puaut, I. (2005). Safe Measurement-Based WCET Estimation. In *5th International Workshop on Worst-Case Execution Time (WCET) Analysis, July 5, 2005, Palma de Mallorca, Spain*. 21, 22

Di Natale, M. & Sangiovanni-Vincentelli, A. (2010). Moving From Federated to Integrated Architectures in Automotive: The Role of Standards, Methods and Tools. *Proceedings of the IEEE*. 2

Diaz, E., Abella, J., Mezzetti, E., Agirre, I., Azkarate-Askasua, M., Vardanega, T. & Cazorla, F.J. (2016). Mitigating Software-Instrumentation Cache Effects in Measurement-Based Timing Analysis. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, 1:1–1:11. 21

Duranton, M., Black-Schaffer, D., De Bosschere, K. & Maebe, J. (2013). The HiPEAC Vision for Advanced Computing in Horizon 2020. 1

Duranton, M., Black-Schaffer, D., De Bosschere, K. & Maebe, J. (2015). The HiPEAC Vision for Advanced Computing in Horizon 2020. 1

Edelin, G. (2009). Embedded systems at Thales: the ARTEMIS challenges for an industrial group. *ARTIST Summer School*. 135

Edgar, S. & Burns, A. (2001). Statistical Analysis of WCET for Scheduling. In *Proceedings of the 22Nd IEEE Real-Time Systems Symposium*, RTSS '01, 215–, IEEE Computer Society, Washington, DC, USA. 24

Elmqvist, J., Nadjm-Tehrani, S., Forsberg, K. & Nordenbro, S. (2008). Demonstration of a Formal Method for Incremental Qualification of IMA Systems. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 5.D.3–1–5.D.3–8. 6, 174, 176

E.Mezzetti, N.Holsti, A.Colin, G.Bernat & T.Vardanega (2008). Attacking the Sources of Unpredictability in the Instruction Cache Behavior. In *Proceedings of the 16th International Conference on Real-Time and Network Systems (RTNS08)*. 7, 24, 31

Evidence (2012). *Erika Enterprise RTOS*. http://erika.tuxfamily.org/drupal/. 144

Feller, W. (1968). *An Introduction to Probability Theory and Its Applications*, vol. 1. Wiley. 35, 183

Ferdinand, C. & Wilhelm, R. (1999). Efficient and Precise Cache Behavior Prediction for Real-Time Systems. *Real-Time Systems Journal*, **17**, 131–181. 7, 30

Ferdinand, C., Heckmann, R., Langenbach, M., Martin, F., Schmidt, M., Theiling, H., Thesing, S. & Wilhelm, R. (2001). Reliable and Precise WCET Determination for a Real-Life Processor. *First International Workshop on Embedded Software (EMSOFT 2001)*. 7, 30

Fernández, M., Gioiosa, R., Quiñones, E., Fossati, L., Zulianello, M. & Cazorla, F.J. (2012). Assessing the Suitability of the NGMP Multi-Core Processor in the Space Domain. In *ACM international conference on Embedded software (EMSOFT)*. 23

Fernandez, M., Morales, D., Kosmidis, L., Bardizbanyan, A., Broster, I., Hernandez, C., nones, E.Q., Abella, J., Cazorla, F., Machado, P. & Fossati, L. (2017). Probabilistic Timing Analysis on Time-Randomized Platforms for the Space Domain. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition*, DATE '17. 79

Financial Times (2015). *Internet of things drives Intel revenues (14/3/2015)*. http://www.ft.com/cms/s/0/66dc4a3e-e2e8-11e4-bf4b-00144feab7de.html#axzz3sqFFU8D3. 1

Fleming, B. (2011). Microcontroller Units in Automobiles. In *2011 IEEE Vehicular Technology Magazine*, 4–8. 3

Forbes (2015). *Weak Desktop Sales Impact Intel's Q1'15 Earnings, Data Center, IoT & NAND See Double Digit Growth (15/3/2015)*. http://www.forbes.com/sites/greatspeculations/2015/04/15/weak-desktop-sales-impact-intels-q115-earnings-data-center-iot_nand-see-double-digit-growth/. 1

Freescale Semiconductors (2008). P4 Series. P4080 Multicore Processor (White Paper). http://www.freescale.com/files/netcomm/doc/fact_sheet/QorIQ_P4080.pdf. 30

Gaisler, C. (2016). LEON3 Probabilistic Platform. http://www.gaisler.com/index.php/products/processors/leon3. 54, 102, 194

Gardner, M. & Lui, J. (1999). Analyzing Stochastic Fixed-Priority Real-Time Systems. In *the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS99)*, 44–58. 23, 24

Garey, M.R. & Johnson, D.S. (1990). *Computers and Intractability; A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co., New York, NY, USA. 31

Girbal, S., Moretó, M., Grasset, A., Abella, J., Quiñones, E., Cazorla, F.J. & Yehia, S. (2013). On the Convergence of Mainstream and Mission-critical Markets. In *Proceedings of the 50th Annual Design Automation Conference*, DAC '13, 185:1–185:10, ACM, New York, NY, USA. 1

Gloy, N. & Smith, M.D. (1999). Procedure Placement Using Temporal-Ordering Information. *ACM Trans. Program. Lang. Syst.*, **21**, 977–1027. 144

Graham, R.L., Knuth, D.E. & Patashnik, O. (1988). *Concrete Mathematics*. Addison-Wesley, Reading MA. 60, 62

Griffin, D. & Burns, A. (2010). Realism in Statistical Analysis of Worst Case Execution Times. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2011)*, 44–53. 24

Grund, D. (2012). *Static Cache Analysis for Real-time Systems: LRU, FIFO, PLRU*. Druck und Verlag. 148

Gustafsson, J. & Ermedahl, A. (2007). Experiences from Applying WCET Analysis in Industrial Settings. In *ISORC '07. 10th IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing, 2007*, 382–392. 53

Gustafsson, J., Betts, A., Ermedahl, A. & Lisper, B. (2010). The Mälardalen WCET Benchmarks: Past, Present And Future. In B. Lisper, ed., *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2010)*, vol. 15 of *OpenAccess Series in Informatics (OASIcs)*, 136–146, Schloss Dagstuhl–Leibniz-Zentrum fuer Informatik, Dagstuhl, Germany. 32, 37, 166, 185

Hansen, J.P., Hissam, S.A. & Moreno, G.A. (2009). Statistical-Based WCET Estimation and Validation. In *9th International Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009*. 5, 24, 39

Hardy, D. & Puaut, I. (2008). WCET Analysis of Multi-level Non-Inclusive Set-Associative Instruction Caches. In *Proceedings of the 2008 Real-Time Systems Symposium*, RTSS '08, 456–466, IEEE Computer Society, Washington, DC, USA. 7, 30, 100

Heckmann, R., Langenbach, M., Thesing, S. & Wilhelm, R. (2003). The Influence of Processor Architecture on the Design and the Results of WCET Tools. *Proceedings of the IEEE*, **91**, 1038–1054. 16, 20

Hennessy, J. & Patterson, D. (2007). *Computer Architecture: A Quantitative Approach*. Morgan-Kaufmann, 4th edn. 18

Hernandez, C., Abella, J., Cazorla, F.J., Andersson, J. & Gianarro, A. (2015). Towards Making a LEON3 Multicore Compatible with Probabilistic Timing Analysis. In *DASIA 2015 20th Data Systems In Aerospace Conference*. 79, 99

HighTec (2008). *Source code of HighTec GPL software*. HighTec, `http://www.hightec-rt.com/en/downloads/sources.html`. 143

Hoyme, K. & Driscoll, K. (1992). SAFEbus. In *Proceedings IEEE/AIAA 11th Digital Avionics Systems Conference*, 68–73. 2

Huntzicker, S., Dayringer, M., Soprano, J., Weerasinghe, A., Harris, D. & Patil, D. (2008). Energy-Delay Tradeoffs in 32-bit Static Shifter Designs. In *Computer Design, 2008. ICCD 2008. IEEE International Conference on*, 626–632. 67

Infineon (2012). AURIX - TriCore Datasheet. Highly Integrated and Performance Optimized 32-bit Microcontrollers for Automotive and Industrial Applications. 31, 144

International Organization for Standardization (2009). *ISO/DIS 26262. Road Vehicles – Functional Safety*. 5, 19, 26, 117, 137, 174, 176

Jalle, J., Kosmidis, L., Abella, J., Quiñones, E. & Cazorla, F.J. (2014). Bus Designs for Time-probabilistic Multicore Processors. In *Proceedings of the Conference on Design, Automation & Test in Europe*, DATE '14, 50:1–50:6, European Design and Automation Association, 3001 Leuven, Belgium, Belgium. 47, 48

Jeffrey Owens, Delphi Automotive (2015). The Design Innovation that Drives Tomorrow. *Keynote at the Design Automation Conference 2015*. 1

Jim Tung, MathWorks, Inc and James Buczkowski, Ford Motor Company (2014). Delivering Smart Automobiles Through Electronics and Software. *Keynote at the Design Automation Conference 2014*. 3

JPL (2009). JPL Institutional Coding Standard for the C Programming Language. JPL DOCID D-60411, Jet Propulsion Laboratory, CalTech. 137

Kim, Y., Broman, D., Cai, J. & Shrivastava, A. (2014). WCET-aware Dynamic Code Management on Scratchpads for Software-Managed Multicores. In *20th IEEE Real-Time and Embedded Technology and Applications Symposium, RTAS 2014, Berlin, Germany, April 15-17, 2014*, 179–188. 31

Kim, Y., Cai, J., Kim, Y., Lee, K. & Shrivastava, A. (2016). Splitting Functions in Code Management on Scratchpad Memories. In *Proceedings of the 35th International Conference on Computer-Aided Design, ICCAD 2016, Austin, TX, USA, November 7-10, 2016*, 60. 31

Kirner, R. & Puschner, P. (2005). Classification of WCET Analysis Techniques. In *Object-Oriented Real-Time Distributed Computing, 2005. ISORC 2005. Eighth IEEE International Symposium on*, 190–199. 20

Kirner, R. & Puschner, P. (2008). Obstacles in Worst-Case Execution Time Analysis. In *11th IEEE International Symposium on Object Oriented Real-Time Distributed Computing (ISORC) 2008*, 333–339. 175

Kirner, R., Puschner, P. & Wenzel, I. (2004). Measurement-based Worst-Case Execution Time Analysis Using Automatic Test-Data Generation. In *Proceedings of the IEEE Workshop on software technology for future embedded and ubiquitous systems (SEUS'05)*, 7–10. 21

Knutson, C. & Carmichael, S. (2001). Verification and Validation for Embedded Software. In *Embedded System Programming Journal*. 4

Kosmidis, L., Abella, J., Quiñones, E. & Cazorla, F.J. (2013a). A Cache Design for Probabilistically Analysable Real-time Systems. In *Proceedings of the Conference on Design, Automation and Test in Europe*, DATE '13, 513–518, EDA Consortium, San Jose, CA, USA. 9, 163

Kosmidis, L., Abella, J., Quiñones, E. & Cazorla, F. (2013b). Multi-level Unified Caches for Probabilistically Time Analysable Real-Time Systems. In *Real-Time Systems Symposium (RTSS), 2013 IEEE 34th*, 360–371. 9

Kosmidis, L., Curtsinger, C., Quinones, E., Abella, J., Berger, E. & Cazorla, F.J. (2013c). Probabilistic Timing Analysis on Conventional Cache

Designs. In *Design, Automation Test in Europe Conference Exhibition (DATE), 2013*, 603–606. 9, 132

Kosmidis, L., Quiñones, E., Abella, J., Vardanega, T. & Cazorla, F.J. (2013d). 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, Paderborn, Germany, June 19-21, 2013*, 1–8. 10

Kosmidis, L., Vardanega, T., Abella, J., Quiñones, E. & Cazorla, F.J. (2013e). Applying Measurement-Based Probabilistic Timing Analysis to Buffer Resources. *International Workshop On Worst-Case Execution Time Analysis (WCET 2013)*. 9, 46, 50

Kosmidis, L., Abella, J., Quiñones, E. & Cazorla, F.J. (2014a). Efficient Cache Designs for Probabilistically Analysable Real-Time Systems. *IEEE Transactions on Computers*, **63**, 2998–3011. 9

Kosmidis, L., Abella, J., Wartel, F., Quiñones, E., Colin, A. & Cazorla, F.J. (2014b). PUB: Path Upper-Bounding for Measurement-Based Probabilistic Timing Analysis. In *Euromicro Conference on Real-Time Systems (ECRTS-14)*. 10

Kosmidis, L., Quiñones, E., Abella, J., Farrall, G., Wartel, F. & Cazorla, F.J. (2014c). Containing Timing-Related Certification Cost in Automotive Systems Deploying Complex Hardware. In *Proceedings of the 51st Annual Design Automation Conference*, Best Paper Award, DAC '14, 22:1–22:6, ACM, New York, NY, USA. 9

Kosmidis, L., Quiñones, E., Abella, J., Vardanega, T., Broster, I. & Cazorla, F.J. (2014d). Measurement-Based Probabilistic Timing Analysis and Its Impact on Processor Architecture. In *17th Euromicro Conference on Digital System Design, DSD 2014, Verona, Italy, August 27-29, 2014*, 401–410. 9, 92

Kosmidis, L., Compagnin, D., Morales, D., Mezzetti, E., Quiñones, E., Abella, J., Vardanega, T. & Cazorla, F.J. (2016a). Measurement-Based Timing Analysis of the AURIX Caches. In *16th International Workshop on Worst-Case Execution Time Analysis, WCET 2016, July 5, 2016, Toulouse, France*, 9:1–9:11. 144

Kosmidis, L., Quiñones, E., Abella, J., Vardanega, T., Hernández, C., Gianarro, A., Broster, I. & Cazorla, F.J. (2016b). Fitting Proces-

sor Architectures for Measurement-based Probabilistic Timing Analysis. *Microprocessors and Microsystems - Embedded Hardware Design*, **47**, 287–302. 53, 99

Kosmidis, L., Vargas, R., Morales, D., Quiñones, E., Abella, J. & Cazorla, F.J. (2016c). TASA: Toolchain-agnostic Static Software Randomisation for Critical Real-time Systems. In *Proceedings of the 35th International Conference on Computer-Aided Design*, ICCAD '16, 59:1–59:8, ACM, New York, NY, USA. 10

Kosmidis, L., Wartel, F., Morales, D., Abella, J., Broster, I. & Cazorla, F. (2017). Dynamic Software Randomisation: Lessons Learned From an Aerospace Case Study. In *Proceedings of the 2017 Design, Automation & Test in Europe Conference & Exhibition*, DATE '17. 113

Kotz, S. & Nadarajah, S. (2000). *Extreme Value Distributions: Theory and Applications*. World Scientific. 6, 27, 29, 40

Lattner, C. & Adve, V. (2004). LLVM: A Compilation Framework for Lifelong Program Analysis & Transformation. In *Proceedings of the International Symposium on Code Generation and Optimization: Feedback-directed and Runtime Optimization*, CGO '04, 75–, IEEE Computer Society, Washington, DC, USA. 107

Law, S. & Bate, I. (2016). Achieving Appropriate Test Coverage for Reliable Measurement-Based Timing Analysis. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, 189–199. 21, 22

Lehoczky, J.P. (1996). Real-Time Queueing Theory. In *Proceedings of the 17th IEEE Real-Time Systems Symposium*, RTSS '96, 186–, IEEE Computer Society, Washington, DC, USA. 24

Lesage, B., Hardy, D. & Puaut, I. (2009). WCET Analysis of Multi-Level Set-Associative Data Caches. *9th International Workshop on Worst-Case Execution Time (WCET) Analysis*. 7, 30, 79, 100

Li, L., Just, J. & Sekar, R. (2006). Address-Space Randomization for Windows Systems. In *Computer Security Applications Conference, 2006. ACSAC '06. 22nd Annual*, 329–338. 126

Li, Y.T.S. & Malik, S. (1995). Performance Analysis of Embedded Software Using Implicit Path Enumeration. In *Proceedings of the 32Nd Annual ACM/IEEE*

Design Automation Conference, DAC '95, 456–461, ACM, New York, NY, USA. 20

Liang, Y. & Mitra, T. (2008). Cache Modeling in Probabilistic Execution Time Analysis. In *Proceedings of the 45th Annual Design Automation Conference*, DAC '08, 319–324, ACM, New York, NY, USA. 24, 79, 172

Lima, G., Dias, D. & Barros, E. (2016). Extreme Value Theory for Estimating Task Execution Time Bounds: A Careful Look. In *28th Euromicro Conference on Real-Time Systems, ECRTS 2016, Toulouse, France, July 5-8, 2016*, 200–211. 27

Littlewood, B. & Strigini, L. (1993). Validation of Ultrahigh Dependability for Software-based Systems. *Commun. ACM*, **36**, 69–80. 4

Lu, Y., Nolte, T., Bate, I. & Cucu-Grosjean, L. (2012). A Statistical Response-Time Analysis of Real-Time Embedded Systems. In *Real-Time Systems Symposium (RTSS), 2012 IEEE 33rd*, 351–362. 172

Marsaglia, G. & Zaman, A. (1991). A New Class of Random Number Generators. *Annals of Applied Probability*, **1**, 462–480. 66

McFarling, S. (1989). Program Optimization for Instruction Caches. In *Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems*, ASPLOS III, 183–191, ACM, New York, NY, USA. 130, 144

Mezzetti, E. & Vardanega, T. (2010). Towards a Cache-Aware Development of High Integrity Real-Time Systems. In *Embedded and Real-Time Computing Systems and Applications (RTCSA), 2010 IEEE 16th International Conference on*, 329–338. 135

Mezzetti, E. & Vardanega, T. (2011a). Cache Optimisations for LEON Analyses (COLA) Final Report. Tech. Rep. COLA-FR-001-i1r1, ESA/ESTEC. 21

Mezzetti, E. & Vardanega, T. (2011b). On the Industrial Fitness of WCET Analysis. In *In Proceedings of the 11th International Workshop on Worst-Case Execution Time Analysis (WCET 2011)*. 4, 19, 20, 22, 31, 100, 175

Mezzetti, E., Ziccardi, M., Vardanega, T., Abella, J., Quiñones, E. & Cazorla, F.J. (2015). Randomized Caches Can Be Pretty Useful to Hard Real-Time Systems. *Leibniz Transactions on Embedded Systems*, **2**, 01–1–01:10. 27

Mezzetti, E., Fernandez, M., Bardizbanyan, A., Agirre, I., Abella, J., Vardanega, T. & Cazorla, F.J. (2017). EPC Enacted: Integration in an Industrial Toolbox and Use Against a Railway Application. In *23rd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS)*. 172

Michael, J.B., Drusinsky, D., Otani, T.W. & Shing, M.T. (2011). Verification and Validation for Trustworthy Software Systems. *IEEE Software*, **28**, 86–92. 4

Mike, J.C.H.P. & Schlansker (1991). On Predicated Execution. Tech. rep., Hewlett Packard. HPL-91-58. 171

MISRA (2013). *Guidelines for the Use of the C Language in Critical Systems*. 137

Mueller, F. (2000). Timing Analysis for Instruction Caches. *Real-Time Syst.*, **18**, 217–247. 7, 30

Mueller, F. & Harmon, D.B.W.M. (1993). Predicting Instruction Cache Behavior. In *In ACM SIGPLAN Workshop on Language, Compiler, and Tool Support for Real-Time Systems*. 7, 30

Muralimanohar, N., Balasubramonian, R. & Jouppi, N. (2009). CACTI 6.0: A Tool to Understand Large Caches. *HP Tech Report HPL-2009-85*. 68, 76

Mytkowicz, T., Diwan, A., Hauswirth, M. & Sweeney, P.F. (2009). Producing Wrong Data Without Doing Anything Obviously Wrong! In *Proceedings of the 14th ASPLOS*, 265–276. 25, 114, 144

Nowotsch, J., Paulitsch, M., Buhler, D., Theiling, H., Wegener, S. & Schmidt, M. (2014). Multi-core Interference-Sensitive WCET Analysis Leveraging Runtime Resource Capacity Enforcement. In *Real-Time Systems (ECRTS), 2014 26th Euromicro Conference on*, 109–118. 4, 20

NVIDIA (2009). NVIDIA's Next Generation CUDA Computer Architecture: Fermi, White Paper. 31

Oka, M. & Suzuoki, M. (1999). Designing and Programming the Emotion Engine. *IEEE Micro*, **19**, 20–28. 31

Paolieri, M., Quiñones, E., Cazorla, F.J., Bernat, G. & Valero, M. (2009a). Hardware Support for WCET Analysis of Hard Real-Time Multicore Systems. In *36th Annual International Symposium on Computer Architecture (ISCA-09)*. 48, 95

Paolieri, M., Quiñones, E., Cazorla, F.J. & Valero, M. (2009b). An Analyzable Memory Controller for Hard Real-Time CMPs. *Embedded Systems Letters*, **1**, 86–90. 48

Pellizzoni, R., Schranzhofer, A., Chen, J.J., Caccamo, M. & Thiele, L. (2010). Worst Case Delay Analysis for Memory Interference in Multicore Systems. In *DATE*. 17

Pelton, S.L. & Scarbrough, K.D. (1997). Boeing Systems Engineering Experiences from the 777 AIMS Program. *IEEE Transactions on Aerospace and Electronic Systems*, **33**, 642–648. 2

Poovey, J. (2007). *Characterization of the EEMBC Benchmark Suite*. North Carolina State University. 32, 37, 50, 110, 136, 166

Potocki de Montalk, J.P. (1991). Computer Software in Civil Aircraft. In *IEEE/AIAA 10th Digital Avionics Systems Conference*, 324–330. 3

Pouillon, N., Becoulet, A., de Mello, A., Pecheux, F. & Greiner, A. (2009). A Generic Instruction Set Simulator API for Timed and Untimed Simulation and Debug of MP2-SoCs. In *Rapid System Prototyping, 2009. RSP '09. IEEE/IFIP International Symposium on*, 116–122, `http://www.soclib.fr/trac/dev`. 32

Prisaznuk, P.J. (1992). Integrated Modular Avionics. In *Proceedings of the IEEE 1992 National Aerospace and Electronics Conference (NAECON)*, 39–45 vol.1. 2

Puaut, I. & Decotigny, D. (2002). Low-Complexity Algorithms for Static Cache Locking in Multitasking Hard Real-Time Systems. In *In IEEE Real-Time Systems Symposium (RTSS)*, 114–123. 31, 79

Puschner, P. (2003). The Single-Path Approach Towards WCET-Analysable Software. In *2003 IEEE International Conference on Industrial Technology,*, vol. 2, 699–704 Vol.2. 171

Puschner, P. (2005). Experiments with WCET-Oriented Programming and the Single-Path Architecture. In *10th IEEE International Workshop on Object-Oriented Real-Time Dependable Systems*. 171

Puschner, P. & Schoeberl, M. (2008). On Composable System Timing, Task Timing, and WCET Analysis. In *Proc. of the 8th Int. Workshop on WCET Analysis*. 6

Puschner, P., Kirner, R. & Pettit, R. (2009). Towards Composable Timing for Real-Time Software. In *Proceedings of the 1st International Workshop on Software Technologies for Future Dependable Distributed Systems, as part of ISORC*. 6, 175

Quiñones, E., Berger, E.D., Bernat, G. & Cazorla, F.J. (2009). Using Randomized Caches in Probabilistic Real-Time Systems. In *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland, July 1-3, 2009*, 129–138. 8, 30

Rapita Systems (2008). *RapiTime*. http://www.RapitaSystems.com/RapiTime. 4, 172

Rebaudengo, M., Reorda, M.S. & Violante, M. (2003). An Accurate Analysis of the Effects of Soft Errors in the Instruction and Data Caches of a Pipelined Microprocessor. In *2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003), 3-7 March 2003, Munich, Germany*, 10602–10607. 40

Reineke, J. (2014). Randomized Caches Considered Harmful in Hard Real-Time Systems. *Leibniz Transactions on Embedded Systems*, **1**, 03:1–03:13. 27

Reineke, J., Wachter, B., Thesing, S., Wilhelm, R., Polian, I., Eisinger, J. & Becker, B. (2006). A Definition and Classification of Timing Anomalies. *International Workshop On Worst-Case Execution Time Analysis (WCET 2006)*. 44

Reineke, J., Grund, D., Berg, C. & Wilhelm, R. (2007). Timing Predictability of Cache Replacement Policies. *Real-Time Systems*, **37**, 99–122. 30, 79

Riley, M., Warnock, J. & Wendel, D. (2007). Cell Broadband Engine Processor: Design and Implementation. *IBM J. Res. Dev.*, **51**, 545–557. 31

Rosen, J., Andrei, A., Eles, P. & Peng, Z. (2007). Bus Access Optimization for Predictable Implementation of Real-Time Applications on Multiprocessor Systems-on-Chip. In *RTSS*. 17

RTCA (2005). *DO-297 Integrated Modular Avionics (IMA) Development Guidance and Certification Considerations*. 6, 174, 176

RTCA and EUROCAE (1992). *DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification*. 5, 19, 26, 36, 137

Rukhin, A., Soto, J., Nechvatal, J., Smid, M., Barker, E., Leigh, S., Levenson, M., Vangel, M., Banks, D., Heckert, A., Dray, J. & Vo, S. (2010). A Statistical Test Suite for the Validation of Random Number Generators and Pseudo Random Number Generators for Cryptographic Applications. Special publication 800-22rev1a, US National Institute of Standards and Technology (NIST). 68, 69

SAE (2001). Guidelines and Methods for Conducting the Safety Assessment Process on Civil Airborne Systems and Equipment. *ARP4761*. 176

Santinelli, L., Morio, J., Dufour, G. & Jacquemart, D. (2014). On the Sustainability of the Extreme Value Theory for WCET Estimation. In *14th International Workshop on Worst-Case Execution Time Analysis, WCET 2014, July 8, 2014, Ulm, Germany*, 21–30. 28

Santini, T., Rech, P., Nazar, G.L., Carro, L. & Wagner, F.R. (2014). Reducing Embedded Software Radiation-Induced Failures Through Cache Memories. In *19th IEEE European Test Symposium, ETS 2014, Paderborn, Germany, May 26-30, 2014*, 1–6. 40

Seznec, A. & Bodin, F. (1993). Skewed-Associative Caches. In *Proceedings of the 5th International PARLE Conference on Parallel Architectures and Languages Europe*, PARLE '93, 304–316, Springer-Verlag, London, UK, UK. 79

Shacham, H., Page, M., Pfaff, B., Goh, E.J., Modadugu, N. & Boneh, D. (2004). On the Effectiveness of Address-space Randomization. In *Proceedings of the 11th ACM Conference on Computer and Communications Security*, CCS '04, 298–307, ACM, New York, NY, USA. 145

Slijepcevic, M., Kosmidis, L., Abella, J., Quiñones, E. & Cazorla, F.J. (2014). Time-Analysable Non-Partitioned Shared Caches for Real-Time Multicore Systems. In *51st Annual Design Automation Conference on Design Automation Conference (DAC'14)*. 48

Spanfelner, B., Richter, D., Ebel, S., Wilhelm, U., Branz, W. & Patz, C. (2012). Challenges in Applying the ISO 26262 for Driver Assistance Systems. In *5th Conference Driver Assistance Systems (Tagung Fahrerassistenz)*. 26

Stephenson, Z., Abella, J. & Vardanega, T. (2013). Supporting Industrial Use of Probabilistic Timing Analysis With Explicit Argumentation. In *2013 11th IEEE International Conference on Industrial Informatics (INDIN)*, 734–740. 26, 126

Suhendra, V., Mitra, T., Roychoudhury, A. & Chen, T. (2005). WCET Centric Data Allocation to Scratchpad Memory. In *Proceedings of the 26th IEEE International Real-Time Systems Symposium*, RTSS '05, 223–232, IEEE Computer Society, Washington, DC, USA. 31

Suhendra, V., Raghavan, C. & Mitra, T. (2006). Integrated Scratchpad Memory Optimization and Task Scheduling for MPSoC Architectures. In *Proceedings of the 2006 International Conference on Compilers, Architecture and Synthesis for Embedded Systems*, CASES '06, 401–410, ACM, New York, NY, USA. 31

Thesing, S., Souyris, J., Heckmann, R., Randimbivololona, F., Langenbach, M., Wilhelm, R. & Ferdinand, C. (2003). An Abstract Interpretation-Based Timing Validation of Hard Real-Time Avionics Software. In *Dependable Systems and Networks, 2003. Proceedings. 2003 International Conference on*, 625–632. 20

Thiele, L. & Wilhelm, R. (2004). Design for Timing Predictability. *Real-Time Syst.*, **28**, 157–177. 4, 20

Tia, T., Deng, Z., Shankar, M., Storch, M., Sun, J., Wu, L. & Liu, J. (1995). Probabilistic Performance Guarantee for Real-Time Tasks with Varying Computation Times. In *the 2nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS95)*, 164–174. 23, 24

Tool Interface Standard(TIS) (1995). *Executable and Linking Format (ELF) Specification*. 129

Topham, N. & González, A. (1999). Randomized Cache Placement for Eliminating Conflicts. *IEEE Trans. Comput.*, **48**, 185–192. 79

Vardanega, T., Bernat, G., Colin, A., Estevez, J., Garcia, G., Moreno, C. & Holsti, N. (2007). PEAL Final Report. Tech. Rep. PEAL-FR-001, ESA/ESTEC. 7

Vera, X., Lisper, B. & Xue, J. (2003). Data Cache Locking for Higher Program Predictability. In *Proceedings of the 2003 ACM SIGMETRICS International Conference on Measurement and Modeling of Computer Systems*, SIGMETRICS '03, 272–282, ACM, New York, NY, USA. 31

Wang, H., Deng, J., Yu, M. & Hei, Y. (2012). A Compact and Robust MCU for Automotive Body Applications. In *2012 IEEE 11th International Conference on Solid-State and Integrated Circuit Technology*, 1–3. 3

Wartel, F., Kosmidis, L., Lo, C., Triquet, B., Quiñones, E., Abella, J., Gogonel, A., Baldovin, A., Mezzetti, E., Cucu, L., Vardanega, T. & Cazorla, F.J. (2013). Measurement-Based Probabilistic Timing Analysis: Lessons from an Integrated-Modular Avionics Case Study. In *8th IEEE International Symposium on Industrial Embedded Systems, SIES 2013, Porto, Portugal, June 19-21, 2013*, 241–248. 4, 8, 23, 32, 54, 61, 78, 121, 122, 123, 125, 139

Wartel, F., Kosmidis, L., Gogonel, A., Baldovin, A., Stephenson, Z., Triquet, B., Quiñones, E., Lo, C., Mezzetti, E., Broster, I., Abella, J., Cucu-Grosjean, L., Vardanega, T. & Cazorla, F.J. (2015). Timing Analysis of an Avionics Case Study on Complex Hardware/Software Platforms. In *Proceedings of the 2015 Design, Automation & Test in Europe Conference & Exhibition*, DATE '15, 397–402, EDA Consortium, San Jose, CA, USA. 100, 112, 113, 114, 139

Watkins, C. & Walter, R. (2007). Transitioning from Federated Avionics Architectures to Integrated Modular Avionics. *In Proceedings of 26th Digital Avionics Systems Conference. DASC '07*. 2, 178

Wenzel, I., Kirner, R., Puschner, P. & Rieder, B. (2005a). Principles of Timing Anomalies in Superscalar Processors. *Proceedings of the Fifth International Conference on Quality Software*, 295–306. 166

Wenzel, I., Kirner, R., Rieder, B. & Puschner, P. (2005b). Measurement-Based Worst-Case Execution Time Analysis. In *Third IEEE Workshop on Software Technologies for Future Embedded and Ubiquitous Systems (SEUS'05)*, 7–10. 21, 22

Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., G.Bernat, Ferdinand, C., R.Heckmann, Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, G. & Stenström, P. (2008). The Worst-Case Execution-Time Problem Overview of Methods and Survey of Tools. *ACM Transactions on Embedded Computing Systems*, **7**, 1–53. 16, 18, 19, 21, 100, 148, 149

Williams, N. (2005). WCET Measurement Using Modified Path Testing. In *5th International Workshop on Worst-Case Execution Time (WCET) Analysis, July 5, 2005, Palma de Mallorca, Spain*. 21, 22

Wilson, A. & Preyssler, T. (2008). Incremental Certification and Integrated Modular Avionics. In *2008 IEEE/AIAA 27th Digital Avionics Systems Conference*, 1.E.3–1–1.E.3–8. 6, 174, 176

Winstanley, A. (2015). *ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade (23/04/2015)*. https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-increase-100x-in-next_decade.php. 3

Xu, J., Kalbarczyk, Z. & Iyer, R. (2003). Transparent Runtime Randomization for Security. In *Reliable Distributed Systems, 2003. Proceedings. 22nd International Symposium on*, 260–269. 145

Yue, L., Bate, I., Nolte, T. & Cucu-Grosjean, L. (2011). A New Way About Using Statistical Analysis of Worst-Case Execution Times. In *ACM SIGBED Review*. 27

Zhu, H., Hansen, J., Lehoczky, J. & Rajkumar, R. (2002). Optimal partitioning for quantized EDF scheduling. 202 – 213. 24

Ziccardi, M., Mezzetti, E., Vardanega, T., Abella, J. & Cazorla, F.J. (2015). EPC: Extended Path Coverage for Measurement-Based Probabilistic Timing Analysis. In *Real-Time Systems Symposium (RTSS) 2015*, 338–349. 172