



JAUME ABELLA FERRER

Nom i cognoms i signatura
del/de la investigador/a

Vist i plau del/de la responsable de la
sol·licitud

Resum del projecte: cal adjuntar dos resums del document, l'un en anglès i l'altre en la llengua del document, on s'esmenti la durada de l'acció

Resum en la llengua del projecte (màxim 300 paraules)

Els sistemes encastats de temps real per funcions crítiques (sistemes CRTE) necessiten estimacions del pitjor temps d'execució (WCET) segures i acurades. Malauradament, els sistemes CRTE necessiten cada cop més rendiment per executar les tasques crítiques. Aquest rendiment només es pot obtenir mitjançant l'ús de memòries cau i processadors multi-nucli, que són molt més difícils d'analitzar des de la perspectiva del WCET. Per tant, tot i que aquestes tècniques proporcionen més rendiment, les tècniques d'anàlisi del WCET no poden obtenir estimacions del WCET acurades. De fet, aquestes estimacions són, en general, pitjors que les que s'obtenen amb processadors molt menys potents. La principal raó és que el comportament del processador és determinista però desconegut i, per tant, en molts casos s'ha d'assumir el pitjor escenari, cosa que dona estimacions del WCET molt elevades.

L'objectiu d'aquest projecte és desenvolupar nous dissenys del maquinari i noves eines d'anàlisi del WCET capaços de generar estimacions del WCET segures i acurades. Amb aquest propòsit el que es farà serà modificar el maquinari amb un comportament difícil d'analitzar degut a la manca de suficient informació per tal que es puguin analitzar de forma probabilística. Per tant, tot i que el pitjor escenari no s'eliminarà, se'n podrà quantificar la seva probabilitat de manera que es pugui donar un WCET segur i acurat pel nivell de fiabilitat requerit, que estarà en línia amb el dels altres components del sistema.

Durant el primer any del projecte hem desenvolupat part de la infraestructura d'avaluació així com les tècniques a nivell maquinari per memòries cau. Durant el segon any hem avaluat les tècniques a nivell maquinari, i desenvolupat tècniques a nivell programari.

Resum en anglès (màxim 300 paraules)

Critical real-time embedded (CRTE) systems require safe and tight worst-case execution time (WCET) estimations to provide required safety levels and keep costs low. However, CRTE systems require increasing performance to satisfy performance needs of existing and new features. Such performance can be only achieved by means of more aggressive hardware architectures, which are much harder to analyze from a WCET perspective. The main features considered include cache memories and multi-core processors. Thus, although such features provide higher performance, current WCET analysis methods are unable to provide tight WCET estimations. In fact, WCET estimations become worse than for simpler and less powerful hardware. The main reason is the fact that hardware behavior is deterministic but unknown and, therefore, the worst-case behavior must be assumed most of the time, leading to large WCET estimations.

The purpose of this project is developing new hardware designs together with WCET analysis tools able to provide tight and safe WCET estimations. In order to do so, those pieces of hardware whose behavior is not easily analyzable due to lack of accurate information during WCET analysis will be enhanced to provide a probabilistically analyzable behavior. Thus, even if the worst-case behavior cannot be removed, its probability can be bounded, and hence, a safe and tight WCET can be provided for a particular safety level in line with the safety levels of the remaining components of the system.

During the first year of the project we have developed most of the evaluation infrastructure as well as the techniques hardware techniques to analyze cache memories. During the second year those techniques have been evaluated, and new purely-software techniques have been developed.



2.- Memòria del treball (informe científic sense limitació de paraules). Pot incloure altres fitxers de qualsevol mena, no més grans de 10 MB cadascun d'ells.

Note: This document describes those activities in the PROARTIS project whose main author is Jaume Abella. Other activities part of the project where Jaume Abella had a secondary role or did not participate at all are not included in the report.

Executive Summary

Activities during the second year have focused on developing the theory and designing a probabilistically analysable processor design as well as developing techniques to perform randomisation at software level. In particular, those tasks include (i) the classification of processor resources, (ii) the analysis of each particular resource, (iii) detailed analysis for cache memories, (iv) some software techniques to achieve randomisation levels similar to those achieved by software means. The rest of the document details each one of those topics in more detail. Finally, we detail the state of the project with respect to the initial objectives set up at the beginning of the project.

The report is provided in an external PDF document.



Probabilistically Analysable Real-Time Systems

Contract Number	2009BP-B00260
Author	Jaume Abella Ferrer
Institution	Barcelona Supercomputing Center - Centro Nacional de Supercomputacion (BSC-CNS)
Date	27-April-2012

Contents

1	Introduction to the PROARTIS technology: Vision and Approach	5
1.1	Background	5
1.2	Probabilistically Analysable Real-Time Systems	6
2	Platform Design Guidelines	9
2.1	Towards a Processor Architecture with Probabilistically Analysable Timing behavior	9
2.1.1	Hardware Support for PTA	11
2.1.2	Probabilistically Modelling the Timing Behaviour of Proces- sor Resources	16
2.1.3	Computing the ETP of execution components using multiple time-randomised resources	20
2.1.4	A Taxonomy of Processor Resources	21
2.2	The PROARTIS Random Cache: Random Placement and Random Replacement Policies	26
2.2.1	Timing Behaviour of Random Caches	29
2.2.2	Random Replacement	29
2.2.3	Random Placement	31
2.2.4	Generalization of the Cache Layout Concept	33
2.2.5	Putting All Together: Set-Associative Caches	34
2.2.6	Hardware Implementation	35
2.3	Compiler and Run-Time Support for Randomisation	38
2.3.1	Random Cache Behaviour on Deterministic Caches	39
2.3.2	Software Components and Memory Objects	39
2.3.3	Random Location of Memory Objects	40
2.3.4	Random Software Approach Implementation: Stabilizer	42
3	Evaluation	45
3.1	Experimental Setup	45
3.1.1	Common processor core architecture	45
3.1.2	Cache architecture	46
3.2	Benchmark Suite	46
3.3	Results for Multi-Level Caches	48
3.4	Multi-Level vs Single-Level Caches	50
3.5	Comparison with Static Timing Analysis and Deterministic Archi- tectures	51
4	Conclusions	53

5 Dissemination

55

1

Introduction to the PROARTIS technology: Vision and Approach

The market for Critical Real-Time Embedded Systems (CRTES), which among others includes the avionics and automotive sectors, is experiencing an unprecedented growth, and is expected to continue to steadily grow for the foreseeable future [12]. Let us for instance look at the automotive domain: a state-of-the-art high-end car, which currently embeds up to 70 Electronic Control Units (ECUs), is predicted to embed many more [11] to account for the inclusion of such new increasingly sophisticated functions as Advanced Driver Assistance Systems (ADAS). For CRTES of this kind it is imperative to ensure the timing correctness of system operation: some form of *Worst-Case Execution Time* (WCET) analysis is needed to that end.

The competition on functional value, measured in terms of application services delivered per unit of product faces CRTES industry with rising demands for greater performance, increased computing power, and stricter cost-containment. The latter factor puts pressure on the reduction in the number of processing units and ECUs used in the system, to which industry responds by looking at more powerful processors, with aggressive hardware acceleration features like caches and deep memory hierarchies.

In this evolving scenario, it must be acknowledged that the industrial application of current WCET analysis techniques [38], which accounts for a significant proportion of total verification and validation time and cost of system production, yields far from perfect results. IBM has for example found that 50% of the warranty costs in cars are related to electronics and their embedded software, and that 30% of those costs are related to timing flaws. These instances of incorrect operation cost industry billions of Euros annually [11].

1.1 Background

Current state-of-the-art timing analysis techniques can be broadly classified in several strands [38]: static timing analysis; and measurement-based analysis.

- Measurement-based analysis techniques rely on extensive testing performed on the real system under analysis using stressful, high-coverage input data,

recording the so-called high watermark execution time, i.e. the longest observed execution time; and adding to it an engineering margin to make safety allowances for the unknown. However, the safeness of the engineering margin is extremely difficult – if at all possible – to determine, especially when the system may exhibit discontinuous changes in timing due to pathological cache access patterns or other unanticipated timing behaviour.

- Static timing analysis techniques rely instead on the construction of a cycle-accurate model of the system and a mathematical representation of the application code which makes it possible to determine the timing behaviour on that model. The mathematical representation is then processed with linear programming techniques to determine a safe upper-bound on the WCET, providing stronger guarantees than measurement-based approaches. Static approaches have one important limitation though: they are expensive to carry out owing to the need to acquire exhaustive knowledge of all factors, both hardware and software, that determine the execution history of the program under analysis. Some processor architectures may dramatically increase this cost. Others, possibly subject to intellectual property restrictions or incomplete documentation, may even make it altogether impossible, and construction of the timing model must resort to observations.

In order to appreciate the complexity of acquiring complete knowledge of execution history, consider a cache model with a Least Recently Used (LRU) replacement policy. The accuracy in predicting the hit/miss outcome of a memory access depends on knowing the full sequence and addresses of the previous memory accesses made by the program up to the point of interest, in order to build a complete and correct representation of the cache state. Any reduction of the available knowledge, e.g. when the addresses of some memory accesses are unknown, leads to a rapid degradation of the tightness of the WCET estimation. In fact, partial knowledge can lead to results as inaccurate as those obtained with no information at all.

1.2 Probabilistically Analysable Real-Time Systems

Our approach begins from the observation that the underlying obstacle in all of the approaches enumerated above is the number of dependences present in the execution of the system. The execution time of each instruction is dependent on a number of factors (events), including the state of the performance-enhancing features of the hardware architecture, the state of the peripheral devices in the system, and in SMP architectures, the software which may be running on another core sharing the same system bus and RAM. A consequence of those dependencies is that in order to appropriately track execution time, and hence WCET of a program, detailed knowledge is required of the system and the dependence factors impending on it.

Our strategy to reduce the cost of acquiring knowledge about execution state (and thus dependent on history) of a program-hardware pair required to perform trustworthy analysis is to adopt a hardware/software architecture whose execution timing behaviour eradicates dependence on execution history by construction. In the

next sections we try to quantify the amount of execution history dependence in current architectures and the cost of acquiring the knowledge to feed current timing analysis techniques for the attainment of tight WCET estimations.

One way to achieve the sought independence is via introducing randomisation in the timing behaviour of the hardware and possibly of the software (while the functional behaviour is left unchanged), coupled with suitable probabilistic analysis techniques [13]. An example of such hardware is a cache memory in which, in the event of a miss, the victim is randomly selected, from any position in the cache. We call this unit of eviction/replacement, cache entry. Under this cache configuration, the probability of hit/miss for an access has a small dependence on execution history, in particular, the number of cache misses between the access under consideration and the previous access to the same address. Note that the hit/miss probability is different from the frequency of events. For instance, a memory instruction may have a 50% hit probability if every time it accesses cache we flip a coin and hit if and only if we get heads. Conversely, if the instruction hits and misses alternately, that instruction does not have a 50% hit probability but a 50% hit frequency. This is so because the outcome, and hence the latency, of each access is fully deterministic.

Next we address how randomisation can be provided by means of hardware so that probabilistic timing analysis can be safely used, and present some approaches to achieve similar effects by means of software. We prove that those designs effectively achieve the requirements of probabilistic timing analysis techniques [13].

2

Platform Design Guidelines

The Probabilistic Timing Analysis (PTA) introduces an important requirement on the timing behaviour of the platform [13], i.e. the timing behaviour of processor instructions must have either no dependence on the execution history or dependence that can be characterised probabilistically. This chapter introduces the PROARTIS platform, a *Probabilistic Platform* composed by the processor hardware and those software components that may affect the timing behaviour of processor instructions, i.e the compiler and memory-related run-time libraries, designed to guarantee that this requirement is met. To that end, the PROARTIS platform introduces randomisation at different execution layers, i.e. at hardware and software, as a way to enable the use of PTA.

Concretely, this chapter evaluates which are the mandatory processor features required to fulfil PTA requirements and provides a detailed taxonomy of processor resources. This chapter also presents the PROARTIS random cache, whose timing behaviour fulfils the PTA mandatory features. Finally, a software technique emulating the timing behaviour of the PROARTIS random cache, but on top of a deterministic cache, is also presented.

2.1 Towards a Processor Architecture with Probabilistically Analysable Timing behavior

Probabilistic Timing Analysis (PTA) approach [10, 13] allows to model accurately the execution time of the component under analysis considered (at some level of execution granularity, e.g. an application, a procedure, a basic block) through a probability distribution function. The response time of each component at that level of granularity can thus be assigned a distinct probability of occurrence¹.

This probabilistic timing behaviour can be described by an Execution Time Profile (ETP for short), which is expressed by a pair of vectors $ETP(R_i) = \{\vec{t}_i, \vec{p}_i\}$, R_i being the component under analysis, $\vec{t}_i = \{t_i^1, t_i^2, \dots, t_i^{N_i}\}$ being all the possible execution times of the R_i and $\vec{p}_i = \{p_i^1, p_i^2, \dots, p_i^{N_i}\}$, being the associated probabilities to each execution time, accomplishing $\sum_{j=1}^{N_i} p_i^j = 1$. In other words, the timing

¹True probability is different from frequency. We refer the reader to Section 2.1.2 where we elaborate on this important observation.

vector in the ETP of a component defines a probabilistic distribution function that enumerates all the possible response times that an execution of that component may incur, and assigns, for each response time, the probability of occurrence. Ultimately, PTA techniques seek to build the exceedance function for the program, which – for a given cut-off execution time value – gives the probability that the actual runtime of the program exceeds that value.

The definition of the ETPs imposes an important requirement in its timing behaviour: the execution times observed by the component must fulfil *the independent and identically distributed* (i.i.d.) property:

- Two random variables \mathcal{X} and \mathcal{Y} are *independent* if they describe two events such that the outcome of one event does not have any impact on the outcome of the other.
- A sequence of random variables is *identically distributed* if all random variables have the same probability distribution.

The level of granularity of the timing events of interest to PTA depends on the analysis approach considered. In case of MBPTA the events of interest are the execution time of end-to-end traversals of program paths in which the timing behaviour of each instruction is known to conform with a specific ETP. The problem at this level then requires assuring independence and identical distribution for the timing of those traversals. Details on how to perform MBPTA timing analysis can be found in [13].

Instead, this section considers the processor instruction as the level of granularity to which an ETP is defined, to better understand the mandatory features required by a probabilistic platform. Other authors who studied PTA considered other levels of abstraction: for example, Petters looked at basic blocks in [31]; Burns and Griffin at time bands in [8].

Approach The PROARTIS platform has been designed so every processor instruction can be characterised by a distinct ETP. We build those ETP incrementally. Without loss of generality, we consider the inner functionality of a processor architecture to employ a number of passive resources (e.g cache, buffers, buses, branch predictors), and we assume each processor instruction will use some of those resources in a given order, whether in sequence or in parallel. We design each such resource so that it can be assigned an ETP: to achieve this for all resources, we use time randomisation in *some* of them. A resource whose timing behaviour is not randomised must be safely assigned a local upper-bound that is not exposed to timing anomalies: any such resource will have an ETP containing a single timing value with a probability of 1. The processor architecture determines the way in which the ETP of individual resources must be convolved² to produce the ETP for a processor instruction. The timing events that the ETP of a processor instruction \mathcal{I}_k describes can be regarded as i.i.d. random variables if and only if: (a) \vec{t}_k does not vary with the history of previous execution in that resource; and (b) \vec{p}_k does

²In probability theory, a convolution is a function that, given the probability distribution functions for two independent random variables, computes the probability distribution function of the combined variables.

not vary with the timing outcome of previous instructions in the sequence in which \mathcal{I}_k is used. This condition allows PTA techniques to *ignore* the internal features of the processor that enable the i.i.d. property to hold for the timing of individual processor instructions. PTA can thus concentrate solely on the program, resting on the knowledge that there will be no effects on the runtime of the program stemming from the history of execution within the processor.

Contribution This section shows, for the first time, a clear understanding of the features that a processor architecture must possess to fit the assumptions of probabilistic timing analysis in the form described above. This section also offers insight into the implementation costs that may be incurred by actually providing them. In this section: (i) We show which architectural changes are required in a processor to meet the PTA requirements by design, regardless of the structure of the program under analysis. Hence, with our by-design *PTA-friendly* processor architecture, no requirements are placed on the program to make it tractable with PTA techniques³. We show how the ETP of resource groups inside the processor can be determined from the ETP of individual resources. We also show that such groups may comprise both time-randomised and non-time-randomised resources. (ii) We present an example of a pipelined processor architecture which incorporates high-performance resources, such as caches and TLB, and yet is provably amenable to PTA. (iii) We classify processor resources identifying how they must be modified to meet PTA requirements. (iv) In section 3 we show that our example processor passes both independence and identical distribution tests.

2.1.1 Hardware Support for PTA

On PTA and its assumptions on the hardware

In the introduction section and in [10] we showed that two PTA variants exist: static PTA (SPTA), and measurement-based PTA (MBPTA). SPTA derives a-priori probabilities from a model of the system. MBPTA derives probabilities from end-to-end runs of the program on the target hardware.

Given a program and a processor, SPTA builds the exact probability distribution for the execution time of the program. SPTA needs to know the ETP for each execution component, building upward from a given level of granularity: for example, this may be the processor instruction – which is our choice here –, the basic block, or the subprogram. SPTA requires the i.i.d. hypothesis to hold for all the granularity levels at which ETP are to be built. Ideally the ETP of individual components of execution would hold constant across distinct uses in different sequences which ensures that the timing probabilities which the ETP express are independent of the history of execution. In reality, a PTA-imperfect implementation of a resource at any level of execution granularity may achieve independence but be unable to guarantee identical distribution. This happens when the timing vector of the ETP of that resource is insensitive to history of execution within the resource itself, but the corresponding probability vector is affected by preceding execution outside of the resource. If this hindrance occurs in the construction of the ETP for

³Of course, a well-structured, timing-aware program makes the quest for flow facts and code-level analysis in general much simpler.

a given processor instruction, then that instruction has one ETP for any distinct sequence of instructions (i.e., a program path) which precedes it. SPTA models one sequence of instructions at a time: individual paths must be combined by taking a pessimistic upper bound. This means that globally non-identical distributions simply add the requirement that the ETP is recomputed for each path, restoring i.i.d. within that sequence. If there are further dependences, e.g. between the outcome of an earlier timing event and the current instruction, then a compound distribution might be computed which considers all the possible outcomes, reducing the granularity to short sequences of instructions. This kind of entanglement quickly spreads however, as evidenced in conventional static timing analysis, and it is highly desirable to maintain strict independence within a sequence to avoid pessimistic simplifications becoming necessary.

MBPTA, as with any other measurement-based approach, instead uses observations to infer an approximation of the program timing behaviour, which must be proven to bound the actual distribution from above at all the points of interest. In MBPTA, since actual measurements *capture* (as opposed to predict) the outcome of the events that make the execution time vary, the low-level ETP need not be known by the MBPTA analysis tool as with SPTA. The fact that ETP exist by construction guarantees that the processor timing behaviour acts as a die with an arbitrarily large number of faces – each representing one end-to-end execution time – and distinct probabilities of occurrence for each face. From this knowledge, and a sufficiently large number of i.i.d. observations, we can use techniques such as Extreme Value Theory [13, 23] (EVT) to derive probabilistic WCET estimations. EVT is a branch of statistics which models the convergence of the tail of a sequence of i.i.d. random variables with a known distribution. When those variables describe execution times – at the required level of granularity – then the tails of sequences of those variables describe the worst-case behaviour of the program. Several researchers have shown how to use EVT with the intent of providing probabilistic WCET estimations [10, 13, 15, 20, 21]. In fact, we have shown how PTA-friendly architectures can be analysed by means of EVT [13].

SPTA and MBPTA both require timing behaviour to be modelled with i.i.d events, although at different levels of granularity. Our PTA approach meets the requirements of both SPTA and MBPTA. In fact, the key is that the ETP of processor instructions *can* be constructed. We consider processor instructions to use a given set of resources in some order. We design those resources so that they can be assigned an ETP for which the i.i.d. hypothesis holds by construction. Resources can be assigned an ETP under two conditions: they are either forced have a constant response time, or their timing behaviour is randomised in a manner that satisfies the i.i.d. hypothesis. We then know we can convolve the ETP of those resources to determine the ETP of the processor instruction that uses them, in accordance with the architectural style (e.g., series or parallel) determined by the processor design. The convolution process and the processor architecture must preserve the i.i.d. hypothesis upward, from processor resources to processor instructions. We show that in our approach this is guaranteed, as well as seeing that it is not necessary for all resources to be time randomised.

In the following we discuss the mandatory features that a processor architecture must exhibit to fit our requirements.

Mandatory Features

We argue that the processor architecture must possess certain ‘mandatory features’, denoted $\{M_1, \dots, M_n\}$, for the founding PTA hypotheses to hold in the end-to-end measurement runs for MBPTA, or in the ETP generation for SPTA. A processor architecture is said to be probabilistically analysable in the time domain, or PTA-friendly, if it provably provides all of those features. As a matter of fact the mandatory features we require are not independent of one another as the satisfaction of some of them enables others to be met. Figure 2.1 depicts the precedence relation among them. We first present them in the natural flow of narrative, which starts with the most dependent, and thus most critical; subsequently we discuss their satisfaction following the precedence relation among them.

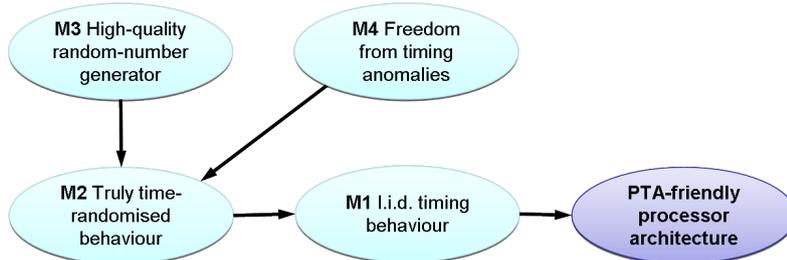


Figure 2.1: The mandatory features to be exhibited by PTA-friendly architectures and the precedence relation among them

- [**M₁**] The hardware must allow the timing events of interest to be modelled as i.i.d. random variables. We view this requirement to be best met at the granularity of processor instructions. One way to achieve this property is to randomise the timing behaviour of certain processor resources, which we require with mandatory feature **M₂** below.
- [**M₂**] The timing behaviour of the resources employed at the level of granularity set for mandatory feature **M₁** must either be truly randomised or allow bounding from above.

In this paper we do not discuss how to randomise the timing behaviour of every resource needed to implement all the instructions in a processor: we claim it can be done, and observe that it has in fact been proposed for such resources as caches, deployed for example in the Aeroflex Gaisler NGMP [2], as well as buses [24]. We further discuss how to incrementally compose resources with either randomised or bounded timing behaviour into subsystems. We accomplish this in a manner which preserves independence for individual instructions whose execution uses multiple resources, as well as identical distribution in the probabilities of execution time for sequences of processor instructions.

- [**M₃**] The hardware must be equipped with a high-quality random number generator that enables processor resources to exhibit time-randomised behaviour at low cost.
- [**M₄**] The hardware must be provably free of timing anomalies [26, 36], because they intrinsically break the independence we assume for the timing behaviour of individual resources. Absence of this feature would prevent the analysis from safely

using the upper-bounding of the execution time of non-time-randomised resources when composing them at a higher level of granularity.

We now discuss the above features in detail and examine the hardware support needed to satisfy them. In the discussion we follow the precedence relation shown in Figure 2.1, from the most basic features, M_3 and M_4 , culminating with M_1 . We differentiate deterministic resources, which may indeed have jittery execution time, from time-randomised resources where timing behaviour can instead be accurately related to a true probability law.

Providing the mandatory features

M_3 : High-Quality Random Number Generator Achieving timing randomisation in processor resources relies on the use of a random number generator. This capability can be efficiently provided by a hardware-based pseudo-random number generator (PRNG) that provides a sequence of numbers. The sequence must have a sufficiently long period so that its output patterns are guaranteed to be extremely unlikely to repeat in the context of the analysis, thereby avoiding correlation between events whose outcome must depend on true probabilities. The degree of randomness attained by the generator can be measured with standard tests such as the one used by the US National Institute of Standards and Technology [35]. The multiply-with-carry PRNG is known to provide the above properties [27] and thus it is our choice. Section 2.2.6 describes an efficient implementation of it.

M_4 : Freedom from timing anomalies Broadly speaking, a timing anomaly is a counter-intuitive timing behaviour of certain hardware resources that may cause existing WCET analysis to yield unsafe estimates if not properly accounted for [26, 34, 36]. A processing resource is a source of timing anomalies when faster execution in the resource may lead to an increase in the execution time of the program overall. This is the case, for example, when a cache miss – the local worst case – results in a shorter execution time than with a cache hit, because of the effect of scheduling decisions in the use of some of the affected resources. The absence of timing anomalies enables us to safely assume the worst-case latency for low-jitter resources.

We want to design a processor architecture in which timing anomalies are excluded by construction. To that end, we refer to [36] which argues that, from a hardware-only perspective, no timing anomalies can ever occur in processors that do not allow resource allocation decisions. A resource allocation is defined as the assignment to hardware resources of either instructions or the micro-operations into which individual instructions can be split on a pipelined architecture. Given a resource allocation, r_i , and a sequence of instructions, we say that a processor does not allow resource allocation decisions if any variation in the latency of any instruction of the sequence, does not change r_i .

M_2 : Achieving truly time-randomised behaviour We regard any concrete resource in a processor architecture as an abstract component that processes requests for access or computation. Each such request has a distinct service time, or latency. The latency of a resource is either fixed or variable. We term jitter the

difference between the best and worst possible latency of any such resource. We classify pre-PTA resources depending upon whether they exhibit jitter or not.

We term *jitterless resources* these processor resources that have a fixed latency, independent of the input request or of the past history of service. Many hardware resources in current processor architectures can be classified as jitterless. Jitterless resources are easy to model for all types of static timing analysis. Building the ETP of a simple instruction that uses a single resource, requires knowing only whether the resource in question is jitterless (information implicit in the instruction) or whether the instruction is part of a sequence of instructions that must incur a delay when using a jitterless resource (information implicit in the architecture). With proper path and pipeline analysis, the types of the resources can be easily determined. Of course, measurements obtained from program runs that only use jitterless resources will perfectly capture their constant impact on execution time. 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 both. Let us look at these dependences in isolation:

- 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, 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 the input request. In this case, the latency is determined by the data carried by the request. For a processor, these data are usually encoded in the instruction that issues the request.
- Although very seldom, there might also be a combined dependence. An instruction may use one execution-dependent and one input-dependent resource. For instance, a load instruction may execute on an architecture in which the latency of the ALU – which all loads use to compute the effective address – depends on the input data given. In this case the latency of the load depends on its input op-code and also on the execution history when it accesses the cache. In this case we consider the latency of the load to depend on the input request as it imposes a greater burden, on a PTA-friendly architecture, than its history-dependent part.

Jittery resources have an intrinsically variable impact on the WCET estimation for a given program. The significance of this impact depends on the magnitude of the jitter, the program under study, and the analysis method. A way to deal with jittery resources in the absence of timing anomalies [34] is to assume that all requests to those resources incur the worst-case latency. This is acceptable if the cumulative impact on the WCET from assuming the worst-case jitter for the resource is deemed low enough by the system designer. We term the jittery resources with suitably low impact, *low-jitter resources*. The problem we must address thus reduces to time-randomising the resources with high jitter: a hard problem, but much smaller in scale than the full processor.

M₁: i.i.d. timing behaviour Jitterless resources have constant latency, hence their timing behaviour is intrinsically independent and identically distributed. For low-jitter resources, we enforce the worst-case latency, so their upper-bounded timing behaviour also becomes i.i.d., so long as mandatory feature **M₄**, absence of timing anomalies, is supported. For high-jitter resources we redesign them so that their timing behaviour depends on random events produced by our PRNG instead of from any other source of variation, be it the initial state or the input data or both.

It is obviously important to determine whether the claim of independence made for a processor architecture can be sustained in reality. Several tests exist for checking whether events are independent: in our experiments we use the Wald-Wolfowitz test [7] that measures whether binary events are biased, or if can be considered random. The identical distribution hypothesis is tested by goodness-of-fit tests, which compare two distributions and provide the degree of likeness. In this work we use the Kolmogorov-Smirnov test [14], which is the one used most often for comparing cumulative distributions, and thus serves our need well.

If all the mandatory features discussed above are fully and completely supported in a processor architecture, then the execution times that are incurred at the level of processor instructions during program runs are guaranteed to have the i.i.d. property required by PTA. In section 2.1.2 we show that we can combine processor resources that meet our requirements in such a manner that the timing events resulting from the execution of individual instructions that use any combination of them can continue to be modelled as i.i.d. random variables. As a result, we can construct ETP for *all* processor instructions and consequently apply SPTA.

MBPTA instead considers events resulting from the observation of end-to-end program runs, at a higher level of granularity than that at which the i.i.d. hypothesis is assured by construction. We must therefore differentiate the MBPTA argument from SPTA. With MBPTA, the property of independence is indeed preserved at the higher level of granularity because no source of dependence can exist between end-to-end runs given that no state is retained in the processor, so long as no logical, software-level, state is allowed to pass between any two such runs. The property of identical distribution must be achieved in the way observations are made, outside of the processor architecture. While PTA-imperfect implementations may prevent an instruction from being considered i.i.d. in isolation from other instructions (as discussed before in this section), observations for one instruction in the context of a preceding sequence of instructions in a PTA-friendly processor indeed are identically distributed. Moreover, observations from outside a sequence of independent instructions (i.e., a program path on a PTA-friendly processor) are identically distributed, too. Observations for a set of paths are identically distributed if and only if they are drawn at random from that same set of paths.

2.1.2 Probabilistically Modelling the Timing Behaviour of Processor Resources

To produce the execution time distribution of a program, in order to determine its probabilistic WCET for a given exceedance threshold, we must combine the ETP of the resources used by the instructions executed by the program. In this section we describe how the ETP for different processor instructions (and in general for any

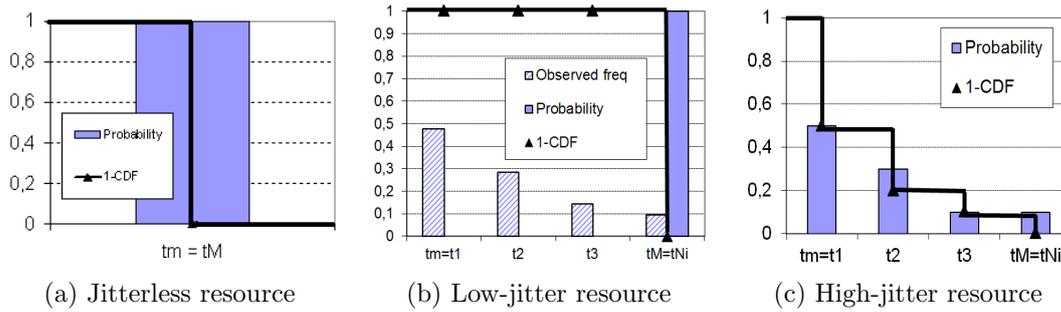


Figure 2.2: Probabilistic timing behaviour of a single instruction for each type of resource

execution component at other levels of granularity) are composed, how those ETP are obtained for some particular arrangements of the resources, and how ETP are obtained in the general case. We show how the probabilistically analysable timing behaviour of each resource and of the program components that use them can be combined into a probabilistically analysable compound behaviour. This enables us to design a complete processor architecture that is PTA-friendly.

Assigning ETP to individual resources

As we have seen above, we only need to randomise the timing behaviour of *high-jitter* resources. Let us examine the effect of doing that.

In order for a resource to be treated with PTA, the probability assigned to each latency in the timing vector of the ETP for that resource must be a *true* probability. We noted in Section 1 that the true probability for a given latency is different to 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 $p_1^1 = 50\%$ of occurrence if – in the implementation of that resource – on every request to it we flipped 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 resource $R_{i=2}$, 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 the outcome, and hence the latency, of each request to that resource are fully deterministic: in this type of resource, information on past events *cannot* be used to provide guarantees about the appearance of future events because they are heavily dependent on the sequence of instructions.

For the purposes of PTA, the timing behaviour of jitterless, low-jitter and high-jitter resources can all be described probabilistically by ETP. Figure 2.2 depicts an ETP for each such type of resource. In each plot we show the relevant latencies of a request to the resource type in question, where $t^1 = t^m$ ⁴ is the minimum latency, $t^M = t^{N_i}$ the maximum latency and $\{t^i\}$ with $i = 2, \dots, N_i - 1$ are latencies such that $t^i \neq t^m$ and $t^i \neq t^M$. For each latency we then show its true probability.

Jitterless resources have an ETP with a single latency with probability of 1 (vertical bar in Figure 2.2(a)). The Probability Distribution Function (PDF) described by the ETP, has a single point at t^m . The Inverse Cumulative Distribution Function (1-CDF) – otherwise known as exceedance function, which is the key product of

⁴When we talk about a single resource we omit the subscript that indicates the resource id.

PTA – also has a single point at t^M with value 0. For jitterless resources, the probability of latency $t > t^m = t^M$ is obviously 0.

Figure 2.2(b) depicts the probabilistic timing behaviour of a low-jitter resource. For the purposes of this discussion, we assume that for this type of resource $t^m \approx t^M$, hence $t^m = t^1 \approx t^2 \approx t^3 \approx t^M = t^{N_i} = t^4$. For these resources we take the worst-case latency instead of randomising the timing behaviour. The stripped bars in Figure 2.2(b) show the *observed frequencies* and not the true probability of each latency. The solid bar shows the safe ETP estimate assumed for resources of this kind. Assuming the worst-case latency may indeed cause pessimism in WCET estimation and thus also in our probabilistic bounding of it, but it also reduces implementation complexity. As for jitterless resources, the probability of latency $t > t^M \approx t^m$ is 0.

Figure 2.2(c) depicts the probabilistic timing behaviour of a high-jitter resource, which we time-randomise. The solid bars show true probabilities. In the figure, without loss of generality, we assume that the ETP of an exemplary time-randomised resource is $\{0.5, 0.3, 0.1, 0.1\}$ for $\{t^m, t^1, t^2, t^M\}$ respectively. Notably, ETP are discontinuous, which reflects the discrete nature of the latencies of the resources we consider. In general, there are no constraints on the latency l_i that a resource may have, other than $l_i \in \mathbb{N}$.

We have seen how the timing behaviour of resources with no, low and high jitter can individually be modelled probabilistically. Later in this section we show how to probabilistically compose the timing behaviour of multiple requests to one resource, as well as for a request that needs to use multiple resources at the same time. These notions help us construct a processor architecture that is probabilistically analysable in the time domain.

Composing the ETP of different resources

One of the properties we gain from randomising the timing behaviour of high-jitter resources is that composite ETP can be easily determined for different program components which use those resources. This calculation is performed by computing the discrete convolution (\otimes) of the discrete probability distributions which describe the latency for each request; this provides a single compound ETP representing the timing behaviour of the combined set of requests. Given the ETP for two resources $R_{i=\{1,2\}}$ in the form of (\vec{t}_i, \vec{p}_i) where \vec{t}_i is the vector of execution times for the resource, and \vec{p}_i the probability for each of those execution times, the convolution of the two ETP is defined as follows: $(\vec{t}_c, \vec{p}_c) = (\vec{t}_1 \otimes \vec{t}_2), (\vec{p}_1 \otimes \vec{p}_2)$ where $\vec{p}_c(n) = \vec{p}_1 \otimes \vec{p}_2(n) = \sum_{k=-\infty}^{\infty} p_1(k) \times p_2(n - k)$ and \vec{t}_c is given by the values of n where $p_c(n) \neq 0$, which follows from the definition of convolution among random variables and probability distribution functions. The resulting latency \vec{t}_c comes from the summation of the contributing latencies being, for all k such that $p_1(k) \times p_2(n - k) \neq 0$, $t_c(n) = t_1(k) + t_2(n - k)$.

The convolution of two ETP requires that the probability of a request taking a given number of cycles is independent of the history of latencies that have occurred in the involved resources, due to previous requests. With our PTA technique, probability distributions of requests that are not independent *cannot* be convolved. In fact, other methods exist (cf. e.g., [18]) which assume full dependence between probability distribution functions, but we do not consider them here.

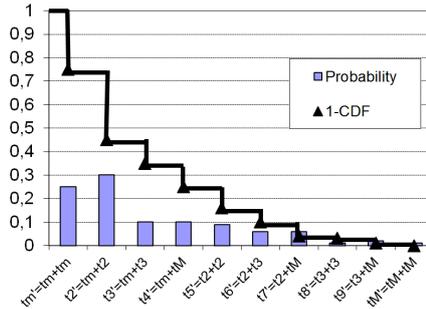


Figure 2.3: Composition of the ETP of two requests using time-randomised resources

ETP reflecting the probabilities of compound requests must be produced to represent all the possible timing interactions in the use of the composed resources. If those ETP should be generated for SPTA, it is imperative to eliminate all architectural dependences. For use with MBPTA, ETP need not be determined so long as they are known to exist at some level of granularity in the execution stack: this will be sufficient for the observations to reflect the effect of true probabilities of each resource for each request.

The convolution of ETP for requests that use jitterless resources leads to the step-like ETP shown in Figure 2.2(a), with the transition from probability 1 to 0 occurring at time instant $t'_m = t_m^1 + t_m^2 = t'_M = t_M^1 + t_M^2$. The case for requests using low-jitter resources, where latency is set to the worst case without time randomisation, is analogous to the transition from probability 1 to 0 occurring at $t'_M = t_M^1 + t_M^2$.

The convolution of the ETP of a single request using one possibly composite time-randomised resource R_1 (with e.g., $ETP_{R_1} = \{\{1, 3, 5\}, \{0.3, 0.4, 0.3\}\}$) together with the ETP of a single request using a jitterless or low-jitter resource R_2 (with e.g., $ETP_{R_2} = \{\{1\}, \{1.0\}\}$) leads to a new ETP whose probability vector \vec{p}_c is the same as for ETP_{R_1} and whose latency vector \vec{t}_c is the one resulting from adding the single latency in ETP_{R_2} to each of the latencies in the timing vector of ETP_{R_1} : for the given values, we thus have $ETP_{R_c} = \{\{2, 4, 6\}, \{0.3, 0.4, 0.3\}\}$. Hence the shape of the CDF of the combined request remains the same as for the request using the time-randomised resource, but with increased indices in the x-axis.

Finally, the architectural composition of multiple requests using time-randomised resources leads to a new ETP where the cardinality of the probability vector is increased up to the product of the cardinality of all probability vectors involved. Figure 2.3 shows the result of convolving two requests using the resource whose ETP is shown in Figure 2.2(c), whose probability vector, without loss of generality, we assumed to be $\{0.5, 0.3, 0.1, 0.1\}$. When the two requests are composed, the probability of the worst-case latency, which happens if both requests take the longest latency, rapidly tends to 0. In this case, and with just two requests, the probability of the worst-case latency is 0.01. With N requests this probability would be 0.1^N .

2.1.3 Computing the ETP of execution components using multiple time-randomised resources

We clarified that the ETP for an individual execution component at the level of execution granularity of interest (a processor instruction in this work) must be statically generated for use with SPTA. Whenever the request uses resources sequentially (Figure 2.4(a)), the relevant ETP can be generated simply by convolving the ETP of the resources involved in the execution, regardless of whether they correspond to time-randomised or non-time-randomised resources. In practice, resources in a processor may be arranged in a variety of ways. A typical arrangement consists of setting them up in parallel, as shown in Figure 2.4(b). Examples of parallel resources are some particular designs of cache memories and translation look-aside buffers (TLB), where cache access and address translation can occur in parallel. In case of parallel arrangements, the probability vector is still obtained by convolution. However, the latency vector, instead of being the addition of the latencies of the two probabilities being convolved, is the maximum latency among those of the probabilities being convolved. This is illustrated with the following example. Let the ETP for resources R_1 and R_2 be $ETP_{R_1} = \{\{1, 4\}, \{0.4, 0.6\}\}$ and $ETP_{R_2} = \{\{2, 3\}, \{0.3, 0.7\}\}$ respectively. The probability vector obtained from their parallel composition in resource R_c is $\{0.12, 0.28, 0.18, 0.42\}$, whereas the latency vector is $\{2, 3, 4, 4\}$, which can be simplified as $ETP_{R_c} = \{\{2, 3, 4\}, \{0.12, 0.28, 0.6\}\}$.

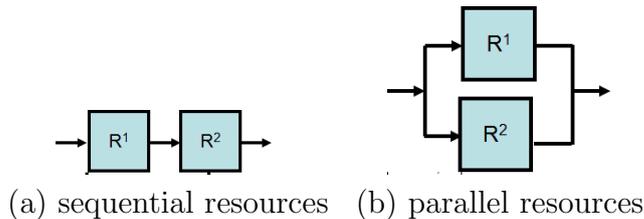


Figure 2.4: Different resource arrangements

In the general case, resources can be arranged in many different manners and multiple requests may be processed simultaneously in different resources. For instance, a pipelined processor may process different requests in different stages and each request may use different sequential or parallel resources in each stage. Moreover, stalls across stages may exist due to resource contention, and those stalls can be difficult to predict a priori if some buffering is in place across stages. This situation may make ETP generation intractable because, in order to generate the ETP, we may have to consider all instructions of the program and all resources of the processor simultaneously. However, as long as all sources of jitter can be described as random events we still can guarantee that each potential execution time occurs with a probability resulting from the combination of multiple random events and, therefore, even if generating the ETP is unaffordable, we *do know* that the ETP of the processor resources exist. SPTA methods may thus indeed become unaffordable for the general case, unless pessimistic assumptions are made. For instance, SPTA may split the execution of the program into small blocks of instructions (e.g., basic blocks, functions, etc.) and generate the ETP by increasing all latencies by the maximum latency that the pipeline may take to empty. Notably, such an assumption is safe in a processor free of timing anomalies, as per our mandatory feature

M_2 . Conversely, MBPTA methods only need to know that ETP exist, but need not generate them. A sufficiently large number of observations made with the precautions discussed at the bottom of Section 2.1.1 will capture all probabilistic events with enough confidence to enable MBPTA methods to obtain accurate bounds on the execution time for those exceedance probabilities of interest.

2.1.4 *A Taxonomy of Processor Resources*

We have shown how some resources may introduce probabilistic jitter that can be easily considered with PTA methods, whereas other resources introduce deterministic jitter that needs to be removed by any means (randomising it, enforcing the worst-case jitter by hardware or enforcing such worst case by the way experiments are designed). However, some other resources introducing jitter may not be easily classified into these two categories. For instance, this is the case of buffers. If a buffer is full it may create a stall that propagates backwards, thus potentially increasing the execution time. However, this particular type of resources is not the source of the jitter but a pure propagator. Therefore, if all jitter occurring in a processor is due to resources producing probabilistic jitter, buffers will get full with a given probability and will simply cause stalls with a given probability. Other resources propagate jitter in a similar way. Although this is the intuition, next we proceed to a more accurate taxonomy of resources so that the the need for randomisation (or lack of it) of the main resources in a processor (yet a single-core processor at this stage of the project) can be understood.

For that purpose we perform an abstract classification of jittery resources based on the source of their jitter. We illustrate such a classification with a particular example of a simple processor. Then the special case of the branch predictor is described in detail and finally, the full taxonomy of resources is provided.

Abstract Classification

We classify the potential sources of jitter into 6 groups depending on the combination of two factors: (i) whether the jitter is produced only by the current event under consideration (no history dependence) or by the combination of previous events and current one (history dependence), and (ii) whether the jitter is deterministic, probabilistic or simply propagated regardless of its source.

Let us first illustrate these four scenarios with some examples before classifying the resources in our processor:

- No history dependence + deterministic jitter. This could be the case of a functional unit whose latency is data dependent. In this particular case we typically enforce the unit to experience always its maximum latency as explained before.
- No history dependence + probabilistic jitter. We do not have any particular realistic example of such a case at this point although some resources might fall into this category in the next phase of the project.
- No history dependence + propagated jitter. In principle such a resource cannot exist because it does not produce any jitter by itself and cannot propagate any jitter if it is history independent.

- History dependence + probabilistic jitter (HD+PJ). This is the typical case of a time randomised cache. The sequence of events between two consecutive accesses to the same data together with the initial cache state, determine the probability vector. If such sequence of events is fixed, then the outcome of each access to this particular resource can be modelled with a random variable, as needed for PTA. We address the meaning of “fixed sequence” below.
- History dependence + deterministic jitter (HD+DJ). This is the case of a cache implementing modulo placement and LRU replacement. Events may experience different latencies based on previous events, but given an initial state and a sequence of events the latency can be just one.
- History dependence + jitter propagation (HD+JP). This is the case of a hardware buffer. A particular instruction may spend a different number of cycles in a buffer depending on previous events. However, as we explain below, given a fixed sequence of events, without data-dependent timing variations and fixed initial conditions, the jitter propagated by this type of resources can also be modelled with random variables.

A *fixed sequence of events* refers to a sequence of instructions, memory accesses, etc. so that exactly the same events occur in the same order. The time that may elapse between two consecutive events may change. In our processor the sequences of events that all history-dependent resources will observe will be fixed given our assumptions: the user provides the relevant paths. That is, once fixed a path of the program under consideration, the processor events observed in that path are the same for every traversal (run) of that path. By definition, the events observed in any subpath of that path are also the same for every traversal. This property is key to ensure that the observed behaviour during the testing phase can be used to derive the behaviour during deployment of the system. This is the case for a cache that observes exactly the same sequence of accesses in all runs (observations) of a particular relevant path.

An Illustrative Example

Let us clarify why HD+JP resources do not violate the i.i.d. properties required for PTA in processor whose all sources of jitter are probabilistic. We start with an example to later extend the discussion to a more complete formulation. For the purpose of this example we assume an architecture with two stages (fetch and execute) that respectively access to time-randomised instruction and data caches (IL1 and DL1 for short). These caches have a probabilistically-modellable timing behaviour. In between both stages there is a 2-entry buffer (see Figure 2.5). In case of hit in both caches and if the buffer is available, an instruction takes 3 cycles: Fetch (F), buffer (b) and Execute (E). The buffer is our HD+JP resource is the focus of this example.

Further assume that we execute a program with four instructions as shown below, whose hit and miss probabilities for each cache are shown next to each instruction. For this example, i_1 always hits in IL1 and has a 0.9 hit probability in DL1. The remaining instructions do not access DL1.

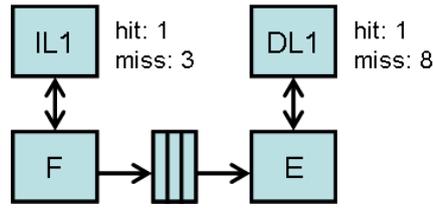


Figure 2.5: Simple processor design used in the example

In this code example, $i1$ may introduce some delay in the execution of the program when accessing DL1: if it misses it will introduce longer delay than if it hits. Note that the IL1 hit probability of $i1$ is 100%. $i2$ and $i3$ may introduce some delay when accessing IL1 only since they are not memory operations.

```

i1: LD (1.0, 0.0); (0.9, 0.1)
i2: ADD (0.7, 0.3); (---)
i3: ADD (0.6, 0.4); (---)
i4: ADD (1.0, 0.0); (---)

```

Next, we depict the 8 different chronograms for each one of the combinations of hits and misses in IL1 and DL1 of all 4 instructions (see Figure 2.6). In particular, we use the vector $\langle DL1i1, IL1i2, IL1i3 \rangle$ for the sake of simplicity in describing the outcome of each DL1 and IL1 access, being H a hit and M a miss. Note that $i1$ and $i4$ have IL1 hit probability of 100% so for this reason $IL1i1$ and $IL1i4$ do not appear in the vector.

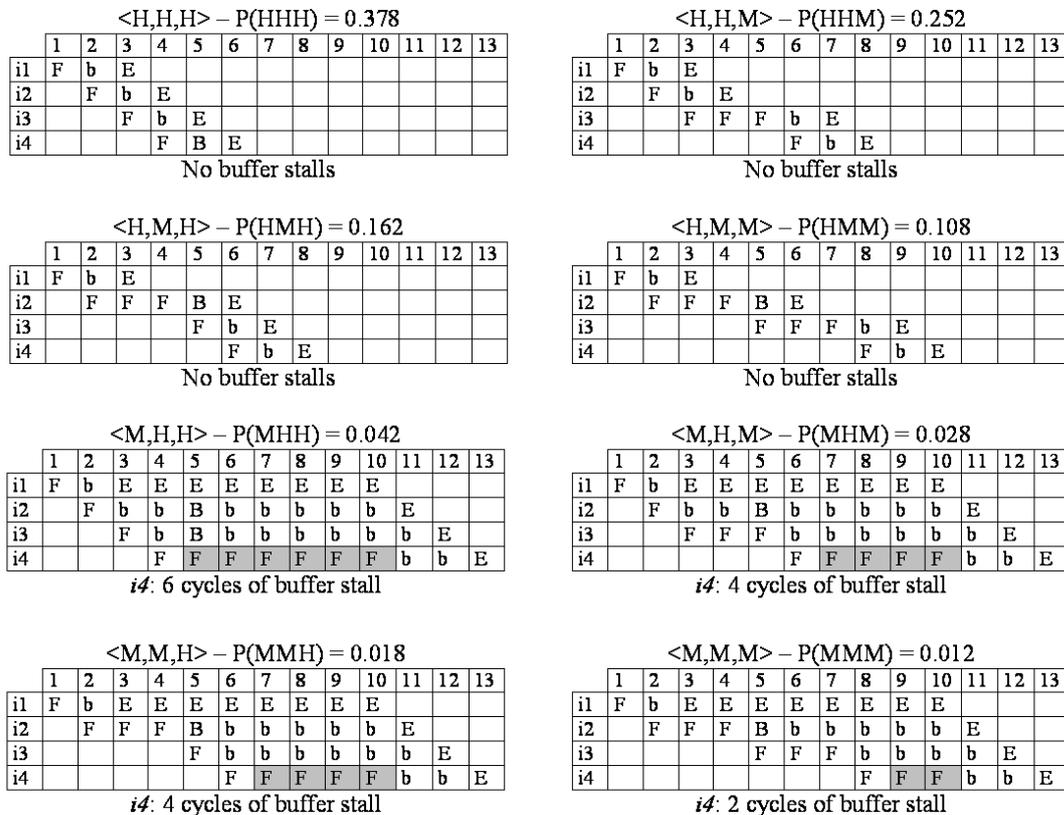


Figure 2.6: Potential chronograms based on the outcome of the different cache accesses

Given a set of fixed initial conditions (empty state of the pipeline) each different combination of probabilistic events (DL1 and IL1 accesses) leads to exactly one fully-deterministic behaviour of the buffer, which is our HD+JP resource. If we compare different outcomes of probabilistic events, we observe that the buffer introduces a different number of stalls (0, 2, 4 or 6 cycles) for each combination of probabilistic events. The number of stalls and the particular cycles in which the stalls occur may repeat in different sequences of outcomes of the probabilistic events occurring (for instance cases $\langle M, H, M \rangle$ and $\langle M, M, H \rangle$ above); however, for a particular sequence of those events the behaviour of the buffer is fully deterministic: all data dependences, which are given by the instructions that are executed and their order, is fully determined by the code being executed. This code is known based on the assumption that relevant paths are provided by the user. Therefore, HD+JP resources (the buffer in our case) cannot create further jitter but simply propagate it or, in other words, given a sequence of outcomes for all probabilistic events the delay of the HD+JP resources is fully deterministic. Therefore, HD+JP resources can affect the duration of the program under each combination of probabilistic events but cannot affect the probability of each combination.

In order to better understand this phenomenon, in Figure 2.7 we show the same example shown before, but describing the processor state in each cycle. As shown, variability in the execution time increases the number of potential probabilistic states that we can reach; however, such variability can only be introduced by probabilistic events. Conversely, buffer stalls (grey boxes) cannot produce such effect and therefore have no effect on the probability of each execution time to occur.

The Case of the Branch Predictor

Branch predictors may introduce jitter because of different sources. The fact that the prediction for the particular branch to be predicted is in the predictor is a probabilistic event if we implement a random placement random replacement predictor. Conversely, once the prediction is found, the fact that the branch is properly predicted is a deterministic event. Such a deterministic event could affect the properties required for PTA; however, this is not the case with the following assumptions:

- Initial conditions are always the same. For instance, the predictor can be reset at the beginning of the execution setting all entries to a predefined value (e.g., branch taken).
- The sequence of events is always the same. Given that relevant paths are given, we know that a relevant path determines which instructions and in which order will be executed. Therefore, the sequence of branches and their outcome is fully deterministic for a given relevant path and cannot introduce variations into the sequence of outcomes provided by the branch predictor.

Thus, if those two assumptions hold (fixed initial conditions and relevant paths given) the only source of variations are random evictions, whose effect can be modelled probabilistically as for any other randomised resource.

Collateral effects. Branch prediction has collateral effects in other resources because in case of a misprediction instructions from the mispredicted path can

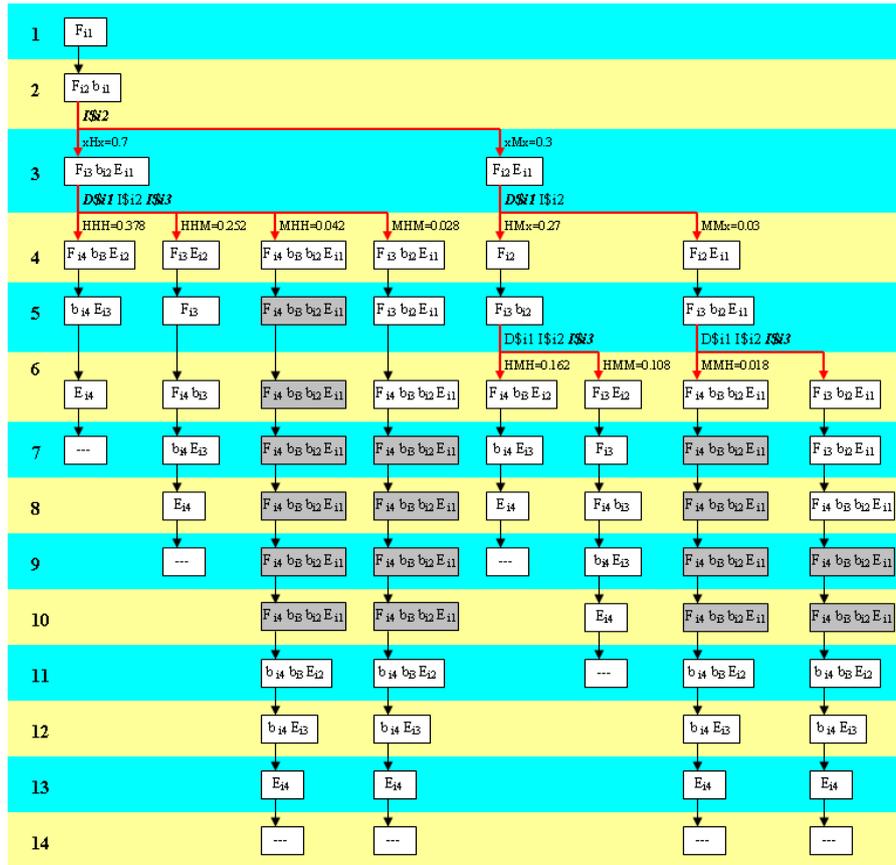


Figure 2.7: Processor Stage Graph

be fetched and partially executed, thus affecting the state of caches and buffers. While this effect is undesirable in general, it occurs with a given probability because the fact that a branch finds its prediction and whether such prediction was properly updated depends only on probabilistic events if all sources of jitter are probabilistic. Therefore, even the effects of fetching and executing instructions from a mispredicted path can be modelled probabilistically.

Even if such effects are probabilistic, one might want to mitigate them to some extent to prevent cache contents to be modified. This could be easily done by stalling cache accesses (and thus the corresponding stages) on a cache miss performed by a speculative access. For instance, if a branch has not been resolved yet and our processor tries to fetch an instruction not present in cache, we can simply stall such operation until the branch is resolved. Once resolved, if it was properly predicted the instruction is fetched and a cache line in the instruction cache is evicted. Otherwise, the instruction is simply discarded together with the other instructions in the mispredicted path. Note that by proceeding this way all speculative accesses hitting in cache are allowed to proceed; however, they do not modify any state in a random placement random replacement cache because no information is kept about history for replacement purposes as in the case of LRU replacement caches.

Hardware Resource Taxonomy

We consider a pipelined processor with in-order fetch, dispatch and retirement of instructions (see Figure 2.8). Fetch and execution stages are equipped with first

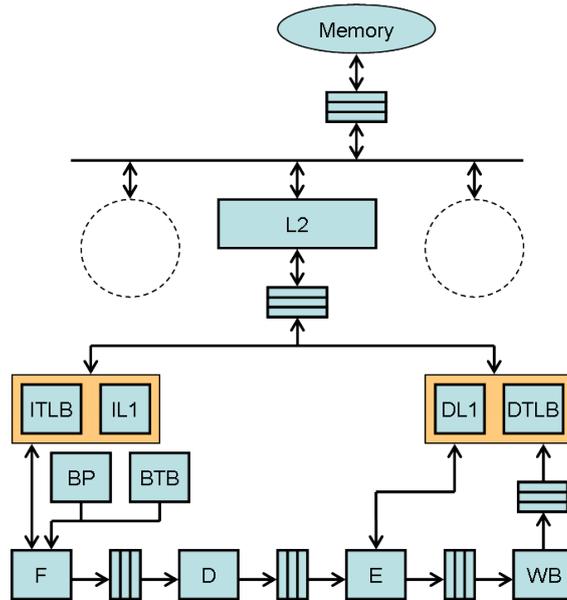


Figure 2.8: Processor components considered within PROARTIS

level instruction and data cache memories respectively (IL1 and DL1 caches for short). Instruction and data translation look-aside buffers (ITLB and DTLB) can exist. Similarly, second-level caches (L2) can be also in place, either specific for instructions/data or shared. Buffers across pipeline stages are deployed to mitigate stalls. Similarly, a store buffer is provided to allow store instructions retire quickly without stalling the pipeline.

Next we list in Table 2.1 the resources with non-fixed latency considered in our architecture, the kind of state they keep and the effect that other software could cause on them.

2.2 The PROARTIS Random Cache: Random Placement and Random Replacement Policies

This section evaluates different cache designs that fulfils the PTA requirements. Caches are of prominent importance hiding the long latency of memory operations, significantly boosting average performance, and it is one of the main focus of PROARTIS.

Caches exemplificates the problems PROARTIS tries to overcome. Providing tight WCET estimations in the presence of caches is hard, since the latency of a particular memory operation strongly depends on the previous memory operations, as well as the cache design itself. Several static WCET analysis methods have been devised to provide WCET estimations for systems with caches [17] [29] [25] [33]. Although those methods are sound and safe, they require detailed knowledge of the sequence of cache accesses in order to provide tight WCET estimations: by keeping the cache state at any point in the program, the analysis can precisely determine whether a memory access will be a hit or a miss. However, modelling all possible cache states is extremely costly, as it requires knowledge of all the memory accesses

Table 2.1: Hardware Resource Taxonomy. *Prob. HD* stands for probabilistic history dependence and *DD HD* stands for data-dependent history dependence

Resource	Prob. HD	DD HD	Influenced by	Details
Instruction cache (IL1, L2, ITLB)	Y	N(*)	All instructions	Prob. HD. All instructions age resources contents. Knowing the number of instructions (or a bound) and the amount of reuse allows accurately considering their impact.
Data cache (DL1, L2, DTLB)	Y	N(*)	Memory instructions	Prob. HD. All load and store instructions age resources contents. As for the instruction cache, the number of such instructions is relevant.
Store buffer	N	Y	All instructions	Function. Data to be stored in the memory system waits in the store buffer until ports are available. DD HD. The state of the store buffer depends on when the last store instructions reached the buffer, which depends also on the instructions executed in between those stores. The worst case is a full store buffer. An upper-bound of the largest latency required to empty the store buffer can be obtained.
Pipeline buffers	N	Y	All instructions	Function. Buffers may exist between pipeline stages to mitigate pipeline stalls. DD HD. All instructions may be stored in those buffers at any time. Their behaviour is analogous to that of the store buffer.
Other buffers	N	Y	All instructions	Function. Buffers may exist anywhere. For instance, between different cache levels, in a memory controller, etc. DD HD. Potentially those buffers could be full when the program is resumed. Upper-bounding the latency to empty those buffers can be done.
Branch predictor	Y	Y(*)	Conditional branches	Function. Predicts whether to jump or not. A table with the prediction is typically used. Prob. HD. All conditional branches age its contents and may replace entries. In that sense behaves as a cache. DD HD. If predictor tables (structures) are tag-less, you cannot check whether an eviction has happened. If the predictor keeps tags identifying the branch they predict, then data (the prediction) cannot be modified by any other instruction other than the instruction that wrote that entry. This is so because any other instruction trying to modify an entry will simply evict it because its tag will be different (it will correspond to a different branch instruction). Otherwise, tag-less predictors may allow data to be modified, but the effect cannot be worse than replacing the entries. Therefore, the probabilistic part is implicitly a bound of the data-dependent part.
Branch target buffer (BTB)	Y	Y(*)	Indirect branches	Function. Some branches need their destination address to be computed dynamically, so a BTB may be used to predict such an address. Prob. HD. Whether the branch finds its prediction in the BTB or not is analogous to the case of the cache. All indirect branches age the BTB. DD HD. Data-dependent modifications of the state can only occur if the interfering software executes part of the code of the relevant path under consideration. In such case the corresponding entry is not replaced but potentially modified. Its worst-case impact is bounded by the probabilistic evictions.
Buses (L1/L2, L1/memory controller, etc.)	Y	N(*)	All instructions	Prob. HD. All instructions can initiate a memory request (e.g., an instruction cache miss). Our architecture will grant bus access in a probabilistic way.
(*) In general, if instructions from the previous piece of software have completed their execution, there cannot be any data-dependent effect. However, in a pipelined processor instructions of consecutive pieces of software can coexist in the pipeline and some stalls can occur if the resource is still in-use. If the different pieces of software are analysed separately their execution times consider the finalisation of the last instruction, so those effects are implicitly captured with the actual execution times.				

performed by the program under study. Moreover, due to the increasing complexity in the software design, the effort to acquire the knowledge about execution history of a program significantly increases [28]. When the required knowledge is not avail-

able, pessimistic assumptions must be made by the analysis. As a result, the net effect is that the WCET estimations in the presence of caches may become overly pessimistic. Therefore, PTA has emerged as an alternative to reduce the amount of information required to perform WCET analysis while providing sound results. Unfortunately, the properties required by PTA analysis are not achieved with current processors due to their deterministic nature, that prevents deriving true probabilities, making the fundamental assumptions required by PTA not to hold. At the cache level, the deterministic behaviour of placement (e.g. modulo) and replacement (e.g. least recently used, LRU) policies makes memory operations depend on the execution history of previous memory accesses in a non-probabilistic way.

This section present several cache analysis designs shows that fulfill these properties, by modelling the timing behaviour of memory operations with probabilities. That is, there must be a distinct probability of each cache access to hit/miss. It is also needed that the timing behaviour of each memory access is independent from previous accesses or this dependence can be measured probabilistically.

The key solution explored in this section is to introduce randomisation in the cache placement and replacement policies. While random replacement has been proposed and used in the past, existing placement functions have a purely deterministic behaviour and thus, they cannot be used in the context of PTA because each potential placement leads to a different behaviour and there is no way to determine the probability of each placement to occur at design time. Hence, we propose a new cache design with random placement that enables the use of affordable set-associative and direct-mapped caches in the context of PTA. The main rationale behind such a design is the fact that the placement function is deterministic during the execution of the program, so cache lookup can be performed analogously to deterministic-placement caches, but placement is randomised across executions by modifying the seed of the parametric hash function used for set placement. This way, each memory access has true hit/miss probabilities and i.i.d. timing behaviour is achieved as needed for PTA. This design also reduces, or even eliminates, dependence across memory accesses, hence reducing the amount of information required by the analysis.

Another objective for our cache design is average performance, which affects key metrics in real-time systems such as energy consumption. Overall, the main contributions of this section are the following:

- A probabilistic analysis of the effect of random replacement and placement policies in the timing behaviour of different cache organisations including direct-mapped, set-associative and fully-associative caches.
- A novel efficient cache design implementing random placement suitable for PTA. It is also combined with random replacement showing how both approaches collaborate synergistically. Our design is proven to have low energy consumption and provides comparable performance to that of conventional modulo placement and LRU replacement set-associative cache designs.
- Hardware-efficient implementations of the parametric hash function used for random placement and a Pseudo Random Number Generator used for random placement and replacement.

2.2.1 Timing Behaviour of Random Caches

Despite its well-known performance benefits, cache memories pose a serious challenge to timing analysis because the latency of a memory request depends on which level of the memory hierarchy the required datum actually resides, which depends on the execution history. That is, because of the temporal and spatial locality characteristics of the memory hierarchy, a memory object resides in cache if: (1) the memory object has been already fetched; and (2) the memory object has not been evicted by another request.

A cache is conceptually a matrix of $S \cdot W$ cache lines⁵ (cells) arranged in S sets (conceptually rows) and W ways (conceptually columns). The set in which a piece of data is placed into cache is determined by the *placement policy*. The placement policy implements a hash function that uses certain bits of the memory address, called *index*, to map each particular cache line into a specific cache set. Since different cache lines can collide into the same cache set, cache sets consist of a given number of lines called *ways*. The size of each cache set is called the *W-way set-associativity* of the cache. The way in which a cache line is placed into a cache set is determined by the *replacement policy*, which selects, among all the ways in a given set, which cache line is evicted to make room for the new cache line. Overall, the timing behaviour of a cache is determined by its placement and replacement policies, and the exact cache state can only be determined if the complete list of memory addresses accessed by the program (which is part of the program's execution history) is known by the analysis.

The use of random placement and replacement allows constructing ETPs that define the probability of hit/miss of each memory request. That is, the timing behavior of every memory access can be defined by the pair of vectors $(\vec{t}, \vec{p}) = \{t_{hit}, t_{miss}\}\{p_{hit}, p_{miss}\}$, where t_{hit} and t_{miss} are the latency of hit and miss respectively and p_{hit} and p_{miss} the associated probability in each case. It is important to remark that the probability of occurrence of a given latency is different from its frequency: while frequency provides information about past events, probabilities enable providing guarantees about the occurrence of future events.

Thus, the existence of the ETPs ensures that the execution times are probabilistic and therefore the system fulfils the i.i.d. property. We provide means to compute the ETP of memory operations when implementing only random replacement (fully-associative caches), only random placement (direct-mapped caches) and a combination of both (set-associative caches).

Finally, in order to implement a random placement and replacement policy that accomplishes with PTA requirements, a proper random number generator is needed. Details on how to implement it are provided later in Section 2.2.6.

2.2.2 Random Replacement

The random replacement (RR) policy must ensure that every time a memory request misses in cache, a way in its 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

⁵To reduce the traffic overhead between the main memory and the cache consecutive memory objects are grouped into blocks called *cache lines*

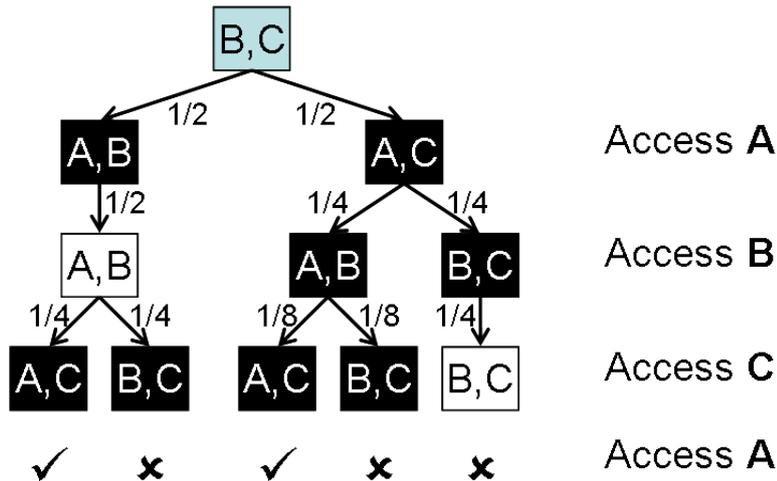


Figure 2.9: Probability tree of the sequence A, B, C, A

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 remaining of this subsection we consider a fully-associative (FA) cache with W ways implementing such a RR policy. Note that FA caches do not use any placement policy as they comprise a single set. As a result, the probability for any cache line to be evicted is $\frac{1}{W}$ on every cache miss.

To develop our argument, let us assume a FA cache with two ways $W = 2$. Further, assume a sequence of $m = 4$ memory accesses A, B, C, A , having $u = 3$ distinct memory addresses mapped to different cache lines, and that initially the cache contains B and C , each one in a different way. Figure 2.9 shows all possible cache states with their associated probabilities after executing 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 $\frac{3}{8}$.

A similar analysis was done in [39] but assuming that each access causes an eviction regardless of whether it hits or misses. Such an approach has shown to simplify the analysis at the expense of lowering hit probabilities artificially and, therefore, increasing the likelihood of high execution times in the execution time distribution obtained. In our case, for a sequence $A, B_1, B_2, \dots, B_x, A$, the probability of the second occurrence of A to be a hit, and so A to survive, can be formulated as follows, where $P\{\overline{B_i \text{ evict } A}\}$ is the probability that B_i does not evict A :

$$P_{hit_A} = P\{\overline{B_1 \text{ evict } A}\} \cdot \dots \cdot P\{\overline{B_x \text{ evict } A}\} \quad (2.1)$$

In which each of the $P\{\overline{B_i \text{ evict } A}\}$ depends on whether B_i is a miss or a hit: A is not evicted if B_i hits. If B_i misses, A is not evicted with a probability of $(W - 1)/W$.

$$P\{\overline{B_i \text{evict} A}\} = P_{hit_{B_i}} + P_{miss_{B_i}} \cdot \frac{W-1}{W} \quad (2.2)$$

The corresponding ETP that models the probabilistic timing behaviour of the cache based on the history of previous memory accesses is the following:

$$ETP(A) = \{t_{hit}, t_{miss}\} \{P_{hit_A}, 1 - P_{hit_A}\} \quad (2.3)$$

2.2.3 Random Placement

The random placement policy we are after has to ensure that the cache set in which a cache line is mapped is randomly selected. As a result, each cache set has exactly the same probability of being selected and this selection is independent of previous ones. Hence, 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. As a result, if the placement policy assigns two memory addresses to the same cache set, they will collide for the whole execution. Instead, the replacement policy has the freedom to allocate the new fetched cache line in any cache way, independently of the memory address. Hence, on every access an associative search is done in all ways of a set, so on evictions the new data can be placed in any way. Instead, the set is determined by the address, so the mapping between the address and the set (placement) has to be fixed.

To deal with this deterministic nature and randomising the timing behaviour of the placement policy, we propose a new parametric hashing function that makes use of a random number generated by a pseudo-random number generator (PRNG), described in Section IV. Our hash function, given a memory address and a random number called *random index identifier* (RII), provides a unique cache set (mapping) for the address that is maintained along the execution. If the RII changes, the cache set in which the address is mapped changes as well. A fundamental property of our proposal is 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. $\frac{1}{S}$. The hardware implementation of our random placement policy is detailed in Section 2.2.6.

Being able to quantify the probability of each memory address to be mapped into a given cache set, and so conflicting with other memory addresses, is fundamental. Given u different memory objects and S cache sets, we define *cache layout* as the resultant mapping of assigning the u memory objects into the S cache lines. Thus, every time the program is executed, the PRNG generates a new RII that leads to a new random mapping function corresponding to a *cache layout*. Different cache layouts cause different cache conflicts among memory addresses, resulting in different execution times. In general, the number of possible cache layouts is given by S^u , where S is the number of sets and u the number of distinct memory addresses.

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 2.2 identifies all possible cache layouts of a program consisting of $u = 3$ memory

Table 2.2: Possible cache layouts for the different accesses of the sequence A, B, C in a idealized random cache with two lines.

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 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 2.2 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 2.2, 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. We call those cache layouts generating different conflicts *cache conflict layouts*. The total number of cache conflict layouts is given by the u^{th} moment of a probability distribution of random permutations [19]:

$$E(X^u) = \sum_{j=1}^S S(u, j) \quad (2.4)$$

where X is the random variable that models the cache behaviour, i.e. the random placement policy, and $S(u, j)$ is the Stirling number of the second kind [9] with parameters u and j . In other words, 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 conflict layouts is 4, identified by a number in parenthesis in the second column of Table 2.2.

With this, we can compute the probability of each cache conflict layout and hence the probability of its resulting execution time for a given program. If we consider the example shown in Table 2.2, 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: $\{(31, 40), (0.25, 0.75)\}$. The cache conflict layouts (1), (2) and (3) lead to an execution time of 40 cycles with an associated probability of 0.25 each. The cache layout (4) leads to an execution time of 31 cycles with an

associated probability of 0.25.

In an arbitrary sequence $A, B_1, B_2, \dots, B_q, A$ 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) 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 entry, the number of cache layouts in which the other q objects are placed in different cache sets is $(S - 1)^q$: the q entries can be placed in all entries except 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 determined by S^{q+1} , the probability of the second occurrence of A being a hit can be computed using the following equation:

$$P_{hit_A} = \frac{(S - 1)^q \cdot S}{S^{q+1}} = \left(\frac{S - 1}{S} \right)^q \quad (2.5)$$

The reuse distance of A , defined as the number of unique addresses (q) between to occurrences of the same memory address, determines how likely it will result in a hit/miss. The higher the q -distance is between two occurrences the less likely is 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, A ($q = 1$).

To sum up, random placement guarantees by design that exists an ETP per memory request based on the history of previous memory accesses, as needed by PTA:

$$ETP(A) = \{t_{hit}, t_{miss}\} \{P_{hit_A}, 1 - P_{hit_A}\} \quad (2.6)$$

2.2.4 Generalization of the Cache Layout Concept

Interestingly, the concept of cache conflict layout also applies to the random replacement policy. However, the random replacement policy does not keep the same cache layout for the whole execution as the random placement policy does (fixing a value for RII): due to the associative search done among all ways in a set on every access, every time a cache line is fetched, it can be randomly mapped into a new cache way, and so, the conflicts among different memory addresses change as well. As a result, a new cache layout is built on every cache miss. This makes the number of cache conflict layouts depend on the number of evictions, which is bounded by the number of memory accesses (m). The total number of cache conflict layouts is given by the m^{th} moment:

$$E(X^m) = \sum_{j=1}^W S(m, j) \quad (2.7)$$

X is the random variable that models the cache behaviour, i.e. the random replacement policy, and $S(m, j)$ is the Stirling number of the second kind with parameters m and j . The m^{th} moment is the number of partitions of m memory accesses into no more than j cache ways.

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 $m = 16$, the number of cache conflict layouts of a DM-RP and a FA-RR caches, considering in both

cases $N = 4$ cache entries ($S = 4$ cache sets in case of the DM and $W = 4$ cache ways in case of the FA cache), is 15 and 178973355 respectively. As expected, the number of cache conflict layouts is much higher for the FA-RR cache than for the DM-RP one because the total number of memory accesses (m) is typically much larger than the number of unique addresses (u). This has an important implication in the worst-case performance of caches. Both caches have the same worst-case cache conflict layout, i.e. the one in which all memory objects are mapped into the same cache entry, resulting in systematic cache misses. However, the probability of experiencing such cache conflict 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.

2.2.5 Putting All Together: Set-Associative Caches

From a performance perspective, Fully-Associative RR (FA-RR) caches are desirable. However, at hardware level such designs may be complex to implement. Instead, DM caches are less complex, though they may have low performance for many programs. Set-associative (SA) caches trade off the benefits of both designs. Broadly speaking, one can argue that, since both random placement and replacement provide i.i.d. execution times, the composition of both approaches, in a set-associative cache, fulfils the i.i.d. property as well. This section proves that set-associative caches combining random placement and replacement also provide the requirements imposed by PTA.

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 the S sets with a probability of $1/S$ while the random replacement policy will evict a way to allocate a new fetched cache line with a probability of⁶ $1/W$:

$$P_{hit_A} = \left(\frac{S-1}{S}\right)^q \cdot (P\{\overline{B_1 evict A}\} \cdot \dots \cdot P\{\overline{B_x evict A}\}) \quad (2.8)$$

Similarly to the computation of the ETP, the number of cache conflict layouts can also be computed as the combination of the number of cache conflict 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:

$$E(X^m) \cdot E(X^u) = \sum_{j=1}^W S(m, j) \cdot \sum_{j=1}^S S(u, j) \quad (2.9)$$

$E(X^u)$ and $E(X^m)$ are the number of cache conflict layouts given by the random placement policy and the random replacement policy respectively. Table 2.3 shows the number of cache conflict layouts of three different cache configurations considering the example of previous subsection: 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

⁶Note that this equation is not valid for direct-mapped and fully-associative caches, whose analytical models are described in equations 2.2 and 2.5 respectively.

Table 2.3: Number of cache conflict layouts for different cache setups

Cache configuration	Number of cache layouts
DM-RP	15
SA-RP+RR	26224
FA-RR	178973355

$m = 16$. All cache configurations are composed of 4 cache lines: a 4-set DM-RP cache, a 2-way 2-set SA-RP+RR cache and a 4-way FA-RR cache.

Overall, 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 conflict layouts increases as the degree of randomness increases as well.

2.2.6 Hardware Implementation

This section describes how to implement both random placement and replacement policies for direct-mapped, set-associative and fully-associative caches as well as the PRNG required for random placement and replacement.

Pseudo-Random Number Generator

A PRNG has to produce a sequence of random numbers that must have sufficiently high level of randomness to ensure that the events we are interested in can be characterised by a true probability. PRNGs use one or more seeds to generate new random numbers and update the seeds themselves. Any PRNG repeats the sequence of numbers whenever the input seed repeats. While this phenomenon is unavoidable in general, it is important that the sequence does not repeat often enough to cause correlation between events whose outcome must depend on true probabilities.

The Multiply-With-Carry (MWC) [27] PRNG satisfies these requirements. The MWC PRNG produces random numbers based on the following set of equations:

$$seed_z = 36969 \cdot (seed_z \& 65535) + (seed_z \gg 16) \quad (2.10)$$

$$seed_w = 18000 \cdot (seed_w \& 65535) + (seed_w \gg 16) \quad (2.11)$$

$$RII = (seed_z \ll 16) + (seed_w \& 65535) \quad (2.12)$$

where $seed_z$ and $seed_w$ are the seeds of the PRNG, $\&$ stands for a logical *AND* function, \gg and \ll stand for logical bit shifts, and RII is the random number generated. Both seeds are updated to produce a different number every time (Equation (2.10) and (2.11)). Recommended initial values for $seed_z$ and $seed_w$ are 362436069 and 521288629 respectively [27].

We provide an efficient implementation of MWC that we call PTA-MWC (Figure 2.10). PTA-MWC comes from the observation that logical *AND*, bit shifts and 16-bit additions required by MWC are simple operations in hardware. Multiplications are much more complex, however they can be transformed into a set of few additions given that one of the operands is known and the number of ones is low. For instance, 36969 (9069h) has only 6 bits set to one, so we can transform such a multiplication into an addition of 6 16-bit numbers. Similarly, 18000

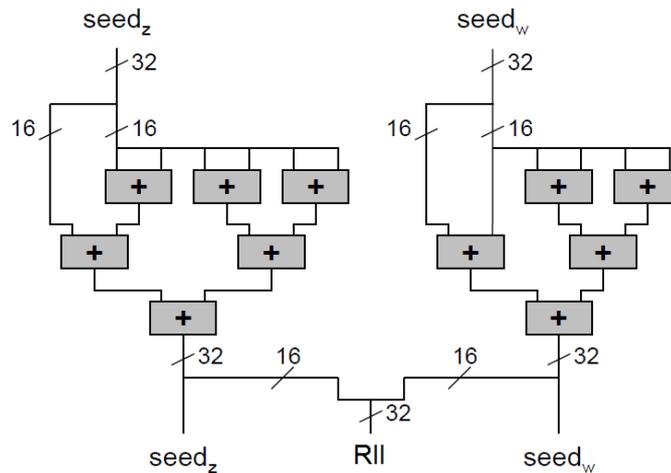


Figure 2.10: Implementation of the PTA-MWC PRNG

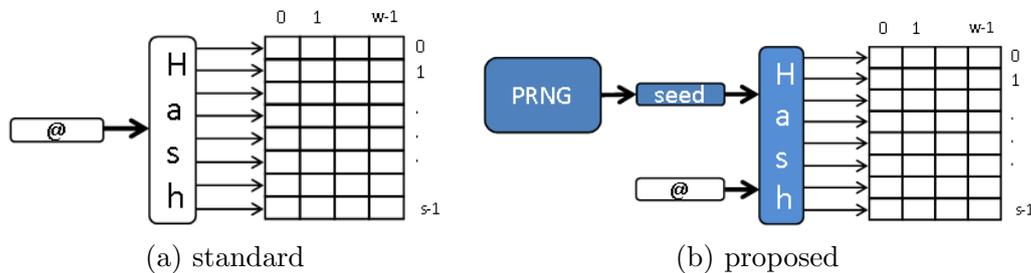


Figure 2.11: Block diagram of the cache design.

(4650h) can be transformed into an addition of 5 16-bit numbers. Thus, each seed generation requires 6 or 7 additions in total, which can be arranged in a binary tree of 3 levels of 2-input adders. The resulting *RII* just selects a subset of the bits of the two seeds, so it does not introduce any delay. The logical design of the PTA-MWC PRNG is depicted in Figure 2.10.

We have implemented the PTA-MWC in the CACTI tool [30], which is an accurate delay, energy and area model for cache memories. PTA-MWC delay and energy are acceptable given that PTA-MWC delay fits within one cycle even for a 4GHz frequency (0.2ns in 65nm technology), and its energy is well below the energy required for a cache read operation (0.8pJ per random number versus 62pJ per read access for a 4KB 16-byte line FA cache), which is a very low overhead given that a random number is only required on an eviction.

The MWC is shown to be one of the highest-quality PRNGs by means of the test battery provided by the US National Institute of Standards and Technology [35]. Those tests evaluate the quality of the bit sequences produced by the PRNGs by studying the distribution of ones and zeros, their patterns, whether subpatterns repeat, etc. The MWC PRNG passes 187 out of the 188 tests proposed (99.5%). Other PRNGs provided together with the test battery have also been studied for comparison purposes and none of them achieved a higher pass rate. The period of MWC is huge (2^{60}). Assuming a processor operating at 1GHz and 1 random number generated per cycle, the random number sequence would take 36 years to repeat.

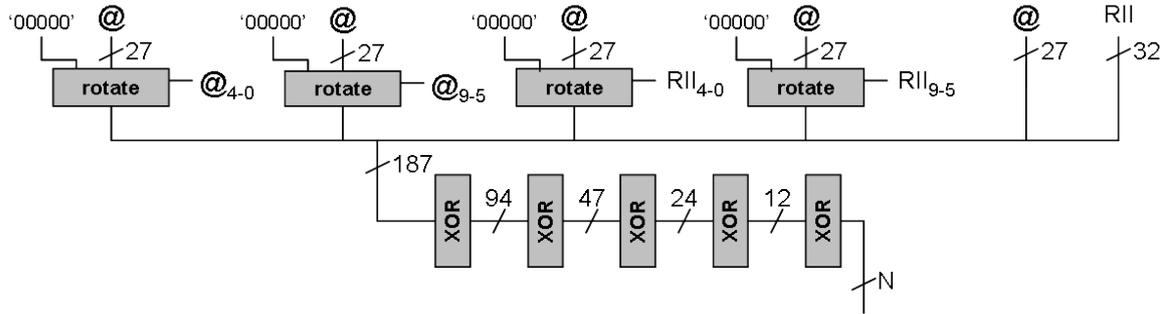


Figure 2.12: Parametric hash function proposed for the RP cache.

Random Replacement Policy

Random replacement policies have been extensively used in various processor architectures, both in the the high-performance and embedded markets. Examples for the latter market are the Aeroflex Gaisler NGMP [3] or some processors of the ARM family [4]. 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. Due to the limitations of hardware designs and the testing procedures, real random number generators are not used in general. Instead, pseudo-random number generators (PRNG) are implemented. We consider the PRNG described in previous Section.

Random Placement Policy

In this section, we propose an implementation of a random placement policy. The key components of this design are (1) a *parametric hash function*, and (2) a low-cost PRNG. The latter has been already described. In fact, RII could be provided by software since it must be updated only once when starting the execution of a program.

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 per se, see Figure 2.11, but some extra logic is added before accessing cache.

The Parametric Hash Function is used to randomise the cache placement. This randomisation is achieved by combining the address of each memory request with a RII in order to create an index for a cache set. This indexing function must satisfy two basic conditions. First, it has to allow each address to be mapped in any of the sets for the different RII values. And second it has to produce different indexes for pairs of addresses when RII changes, to ensure that cache set conflicts among addresses will not repeat systematically. That is, if two addresses conflict in the same set for one run, their probability of conflict for subsequent runs (on which a new RII is generated) is low.

Figure 2.12 shows our implementation of the parametric placement function, which accomplishes both conditions. As mentioned before, the hash function has two inputs, the bits of the address used to access the set (index bits), ‘@’ in the figure, 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 bit) 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 two rightmost rotate blocks of the figure. By doing this, we ensure that when a different *RII* is used, the mapping of that address will be changed. Analogously, the address bits are rotated based on some bits of the address itself. This operation, which is performed by the two leftmost rotate blocks, changes the way that the addresses 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* (187 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.

With this design, by construction, all cache sets have the same probability of being indexed when *RII* is changed. Also the address mapping of different addresses changes in a different and random way. Therefore, conflicts between addresses change across runs as shown in the next section.

As shown in Figure 2.12, the hardware implementation of the hash function consists of 4 rotate blocks and 5 levels of 2 input XOR gates. Each rotate block can be implemented with a 5-level multiplexer [22]. Since 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 placement function into the CACTI tool [30]. Results for several cache configurations show that energy per access grows around 3% 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, given that WCET is mostly influenced by the miss latency.

2.3 Compiler and Run-Time Support for Randomisation

The location at which memory objects of a program are placed into cache, i.e. the *cache layout* (see Section 2.2.4), is determined by the placement and replacement policies. The PROARTIS random cache design implements both random policies making cache conflicts layouts, and so observed execution time, vary randomly across program invocations. This makes the PROARTIS random cache design provide execution time observations fulfilling the i.i.d. property required by the PTA methods.

However, a cache behaviour similar to random caches can be achieved in deterministic cache designs as well if appropriate software support is provided. This section presents a compiler technique that allows deterministic caches to behave as a random cache.

2.3.1 *Random Cache Behaviour on Deterministic Caches*

In deterministic cache designs, e.g. modulo placement or LRU and FIFO replacement policies, given a program and a deterministic cache design, just one cache conflict layout is built, making the execution time not varying across invocations of the program. The reason is that each memory object is mapped into the exact same cache position based on: (1) its memory address, which determines the cache set and (2) its relative order with respect to the other memory objects, which determines the cache way. This is an issue because the placement of the program, system software and dynamic objects will likely change at deployment time, thus invalidating any conclusion obtained at testing time.

Hence, in order to force deterministic cache designs to behave as random caches so that conclusions at testing time hold at deployment time, memory objects must be located into random cache positions. To do so, software must ensure that (1) memory objects are placed in random memory locations and (2) the order in which memory objects are accessed is random. The former will make the cache set to be randomly selected at every new memory location, and so mimicking the behaviour of a random placement policy. The latter will make memory objects to be randomly placed in cache ways, mimicking the behaviour of a random replacement policy. It is important to notice that the deterministic behaviour of the cache does not *break* the random decisions taken by the software. That is, because memory objects are located and accessed randomly, the cache conflict layout that the deterministic cache generates is different at every new memory location and access, maintaining the desired random properties. Unfortunately, the order in which memory objects are accessed depends on the functional behaviour of programs and so changing it would result in an incorrect program execution.

Only software approaches to randomise the placement of memory objects can be applied. The fact of changing the placement of objects does not change the functional behaviour of the program, and only affects its timing behaviour since different cache conflict layouts are generated, which is the desirable effect. This makes the software approach as the perfect candidate to be applied to current high performance processor designs that already implement a random replacement policy [3,5]. Next sections describe in detail software techniques considered within this project.

2.3.2 *Software Components and Memory Objects*

A memory object refers to a memory entity, normally stored in consecutive memory addresses, that can be manipulated by a software or a hardware component. In case of software, memory objects may refer to programs, functions, basic-blocks, data structures, etc. Each of these objects is manipulated by different software components. Next, we list four of the main software components:

- *Compiler*. It is in charge of translating the program source code into object code. Hence, it creates all objects required by a program, placing them into different memory regions, i.e. code and data regions. Current compilers locate program objects into consecutive memory addresses as defined into the source code.
- *Linker*. It is in charge of building the program executable by combining multiple object code files into a single executable, placing code and data

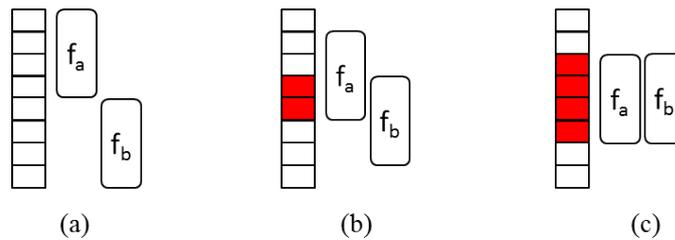


Figure 2.13: Cache location of functions f_a and f_b into a direct-mapped cache implementing modulo placement policy.

memory regions from different object files into a single and unified code and data region.

- *Operating System.* It is in charge of loading the executable program into main memory.
- *Run-time memory-related libraries.* It is in charge of managing dynamic memory services to programs.

In this project we have focused on functions and stack memory objects managed by the compiler.

2.3.3 Random Location of Memory Objects

In order to elaborate our argument, we assume a direct-mapped cache implementing modulo placement policy. Moreover, let us consider a program consisting of a loop that calls two functions, f_a and f_b , each one composed of sequential code. The total size of the two functions is smaller than the cache size and so the effectiveness of the cache will be determined by the cache layout, i.e. where functions are placed in cache.

Figure 2.13 shows three different cache layouts. In Figure 2.13(a), the two functions are placed in consecutive memory positions without colliding one each other and so taking maximum benefit of the cache. However, if they are placed in memory positions such that the modulo function makes two code regions from the two functions collide into the same cache set, the effectiveness of the cache will be lower or even null because systematic cache misses will occur. Figure 2.13(c) shows the case in which f_a and f_b are located in memory addresses that map into the same cache lines. Figure 2.13(b) shows an intermediate case in which half of the functions collide into the same cache lines.

Therefore, forcing f_a and f_b to be randomly located in main memory makes the collision of the two functions into the same cache lines (Figure 2.13(c)) occur with a given probability.

It is important to remark that the random location of memory objects does not remove cache conflicts within memory objects because the relative position of memory addresses of the object remains the same. Hence, cache conflicts among object addresses will remain the same, independently of its memory position. However, such a deterministic behaviour at the object boundary does not affect the i.i.d. properties provided by the random object location, if the size of the object is

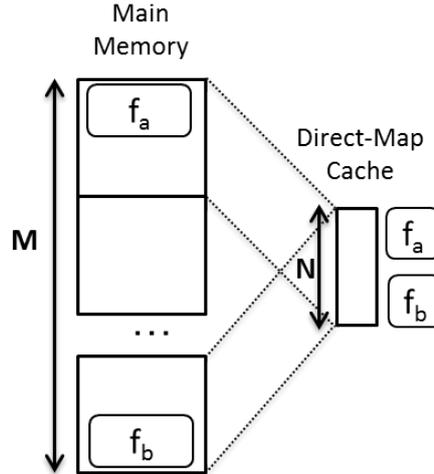


Figure 2.14: Location of functions f_a and f_b into the main memory.

selected properly. Thus, if we consider a program composed of one function (assuming no operating system interaction), the random location will have no effect on the execution time, and so the i.i.d. property will not be fulfilled. Instead, if we consider the same program composed of multiple functions, of a size equal to the cache line size, the random location will make the deterministic cache to behave as implementing a random placement policy.

Influence of the Deterministic Placement Policy

The hardware random placement policy presented in Section 2.2.3 has a probability of $1/S$ of selecting a cache set, S being the total number of cache sets ($S = 8$ in the example above). Hence, in order to mimicking the placement policy, the software approach should select the cache set in which the object is placed with the same probability. By doing so, we can compute the probability of f_a and f_b to collide into the same cache sets, i.e. $1/S \cdot 1/S$. Unfortunately, this probability not only depends on the software methodology but also on the deterministic placement policy considered. Next we discuss about it.

Deterministic placement policies use the index bits of the memory address to identify the cache set. This makes the memory address space to be logically divided into MS different chunks, M being the main memory size. Within each chunk, memory addresses are mapped to the same cache set. Figure 2.14 shows logical memory address space division done by the modulo placement policy and the location in main memory and in cache of functions f_a and f_b . This property guarantees that, if the main memory entry is randomly selected with a probability of $1/M$ its placement into the cache set will fulfil the $1/S$ property as well. That is, because each memory address within a chunk has an associated cache set, the probability of selecting one memory entry within a chunk can be computed as $\frac{M/S}{M} = \frac{1}{S}$.

Unfortunately, the probability of placing a memory object into a given main memory entry is not the same for all objects and it depends on previous locations. In order to elaborate our argument, consider the example of Figure 2.14 in which f_a and f_b occupy four memory entries each. Moreover, let us consider a main memory with $M = 1024$ entries and a direct-mapped cache with $S = 8$ cache sets (which

makes the main memory to be divided into 128 chunks). If f_a is the first object to be randomly located, the probability of it being placed into a given memory entry is $1/M$, and so the probability of selecting a cache set is $128/1024 = 1/8$. However, when f_b is randomly located, this probability changes since it can be only placed in the remaining memory entries, i.e. $M - size(f_a)$, i.e. 1020. As a result, the number of memory entries associated to the cache sets in which f_a is mapped are less (one less in the example). Thus, the cache sets in which f_a is placed will have a probability of $127/1020$ while the rest will have a probability of $128/1020$. In conclusion, those cache sets already mapped are less probable to be selected.

Such an effect can be eliminated if objects are located at chunk granularity. In other words, the remaining memory entries within a chunk are not considered to allocate new object. By doing so, in the previous example f_b is placed into the cache set with a probability of $127/1016 = 1/8$. However, it is also important to remark that the probability bias of cache sets assigned to already occupied memory entries tends to disappear as M becomes much bigger than S (which is commonly the case), and so this effect is negligible. If we consider a case in which $M = 10^6$ and $S = 32$, the probability bias differs in the fifth decimal (0.03125 and 0.031249) of a cache set assigned to a memory entry with respect to another that is not.

2.3.4 *Random Software Approach Implementation: Stabilizer*

We have focused on the random memory location of function and stack frame objects. In both cases, support from compiler and run-time libraries is required. That is, although the compiler is the software component in charge of defining the order at which functions and stack frames are placed within a program, special support to locate functions and stack frames to new random memory positions is required. In this sections we present Stabilizer, a compiler and run-time system that provides the required services at both software component levels to randomly allocate functions and stack frame objects:

- The Stabilizer compiler pass has been developed within the LLVM compiler [1]. Each source code file is first compiled to LLVM bytecode using the `llvmc` compiler driver. The resulting *bytecode* file is then linked and processed with LLVM's optimisation tool running the Stabilizer compiler pass. The resulting executable is then linked with the Stabilizer run-time library which performs the dynamic layout randomisation.
- The Stabilizer run-time library is based of DieHard [6]. DieHard is a memory allocator that uses heap randomization to prevent memory errors, making unlikely that the use of free and out of bounds accesses corrupt live heap data and providing probabilistic security guarantees.

Function Randomisation

The function randomisation technique re-allocates a function by copying its body to a new random memory location. A *Relocation Table* (RT) is placed at the end of each new relocated function to identify the addresses of all globals and functions referenced by the relocated function (see Figure 2.15). By doing this, every function

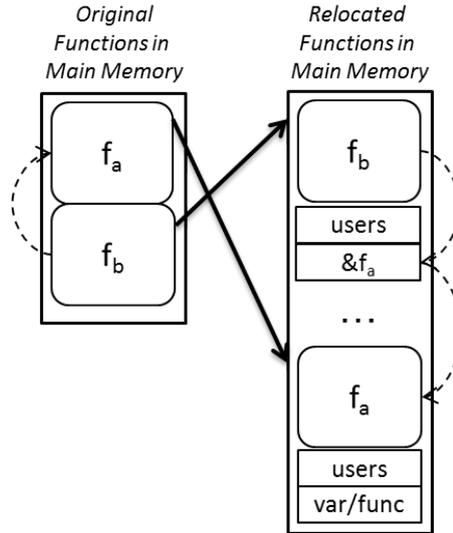


Figure 2.15: Relocation table placed immediately at the beginning of functions f_a and f_b into the main memory.

call or global access in the function is indirected through the RT. Each function refers to its own adjacent relocation table using relative addressing modes, so two randomly located copies of the same function do not share a relocation table. In Figure 2.15, once functions have been relocated f_b calls f_a (dotted line) through the RT. Moreover, the RT contains a *users* counter that tracks the number of active users of the function.

The compiler adds the required code within each function to manage the RT, including code to increase and decrease the *users* field. The random function location is composed of two phases:

- *Initialisation.* During the initialisation phase of the program, the Stabilizer overwrites the first byte of every relocatable function with a software breakpoint. When a function is called, the Stabilizer run-time library intercepts the trap and relocates the function.
- *Random memory location.* Functions are relocated in three stages: First, the Stabilizer run-time library requests a sufficiently large block of memory to the DieHard memory allocator and copies the function body to this location. Second, the function's RT is constructed next to the new function location with the *users* counter set to zero. Finally, Stabilizer overwrites the beginning of the function's original base address with a static jump to the relocated function. Relocation tables are not present in the program binary but are created on demand.

Interestingly, Stabilizer also allows to re-allocate functions on flight while the program is being executed, as the cache random placement policy does when changing the *RII*. To do so, all running threads are interrupted. Stabilizer then processes every function location in the active locations list. The original base of the function is overwritten with a breakpoint instruction, and the function location is added to the locations list.

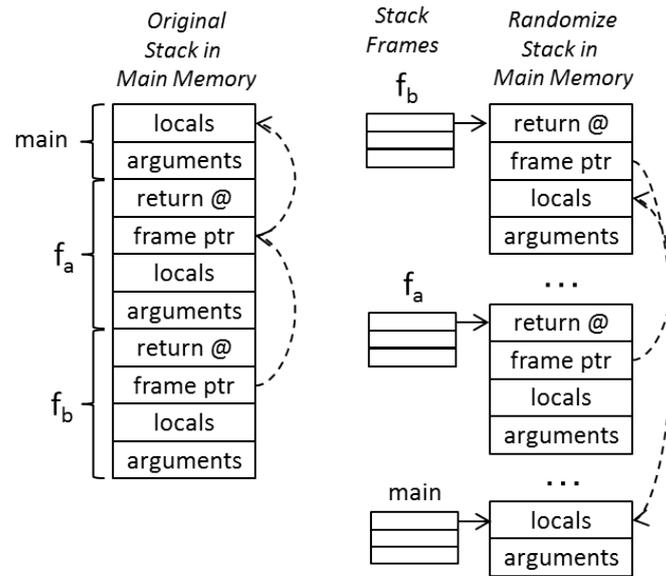


Figure 2.16: Relocation table placed immediately at the beginning of functions f_a and f_b into the main memory.

Stack Randomisation

Stabilizer randomises the stack by making it non-contiguous: each function call moves the stack to a random location. These randomly placed frames are also allocated via DieHard, and Stabilizer reuses them for some time before they are freed. This bounded reuse improves cache utilisation and reduces the number of calls to the allocator while still enabling re-randomisation. Every function has a per-thread depth counter and frame table that maps the depth to the corresponding stack frame. 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.

Similar to the function randomisation, stack randomisation allows applying re-randomisation. To do so, Stabilizer invalidates those saved stack frames by setting a bit in each entry of the frame table. When a function loads its frame from the frame table, it checks this bit. If the bit is set, the old frame is freed and a new one is allocated and stored in the table.

Special handling is required when a stack randomised function calls an external function. Because external functions have not been randomised with Stabilizer, they must run on the default stack to prevent overrunning the randomly located frame. Stabilizer returns the stack pointer to the default stack location just before the call instruction, and returns it to the random frame after the call returns. Calls to functions processed by Stabilizer do not require special handling because these functions will always switch to their randomly allocated frames.

3

Evaluation

In this section we show the evaluation results for our hardware mechanisms (software ones have not been tested yet). In particular the results shown in this section correspond to the hardware design presented in Figure 2.8, which implements our random placement and replacement caches. Those results are compared against the same processor but implementing modulo placement and random replacement caches.

3.1 Experimental Setup

3.1.1 Common processor core architecture

We use a 4-stage pipelined core architecture that operates as follows:

- **Fetch:** Instructions are fetched in-order from the Instruction cache (I\$) into the processor. If the architecture has Instruction Translation Lookaside Buffer (ITLB), it is accessed in parallel to translate pages;
- **Decode:** Instructions are decoded in one cycle. Decoded instructions are issued to the execute stage;
- **Execute:** Instructions are executed in one cycle, except for memory operations. Read operations access the data cache (D\$) during this cycle blocking the execution of further instructions until data are fetched. The Data TLB is accessed in parallel with the D\$;
- **Write-back:** The instructions commit in one cycle. Write operations update a write buffer so they do not block this stage (as long as the buffer is not full).

Instructions are issued in the program order. For in-order processors, no instruction can be issued until all previous instructions have been issued. This is frequently the case for CRTES as this feature facilitates timing analysis. For instance, the LEON4 processor [2], used by the European Space Agency, issues one instruction per cycle, in-order. Similarly, instructions finalise their execution in-order so that no instruction updates the state of the register file or memory until all older instructions have been committed. This commit policy is common in CRTES too. Whereas the cited LEON4 allows instructions to finalise out-of-order, most of them have

a one-cycle latency, hence they typically commit in-order. Conceptually, both in-order and out-of-order finalisation have identical effects on WCET analysis because they do not introduce jitter. We consider in-order finalisation because it simplifies our experimental implementation and also speeds up simulation. However, any other architecture compliant with the requirements defined in Section 2.1.1 could be deployed.

The latency of each instruction is fixed, except for the first stage (fetch) in which the instruction cache is accessed. The latency of the fetch stage depends on whether the access hits or misses in the instruction cache: a hit has 1-cycle latency and a miss has *imisslat* latency that depends on the cache architecture selected. After the decode stage, memory operations access the data cache so they can last 1 or *dmisslat* cycles depending on whether they miss or not. Overall, the possible latencies of a non-memory operation are $(N+1, N+imisslat)$ depending whether it hits or not in the instruction cache, where N is the number of cycles it takes executing the instruction (e.g. integer additions take 1 cycle). Memory operations have 3 possible latencies: 2 cycles (1+1) when it hits both instruction and data caches; $imisslat + dmisslat$ when it misses in both caches; $imisslat + 1$ when it hits only one of both caches (assuming that *imisslat* and *dmisslat* have the same value).

3.1.2 Cache architecture

Both instruction and data cache size is 4-KB with 16-byte line size. Associativities considered are 1-way (direct-mapped), 8-way, 32-way and 256-way (fully-associative). Both caches, instruction and data, implement the random placement and replacement policies presented in Section 2. The random replacement logic implements the multiply-with-carry PRNG [27]. The D\$ is write-through, so that all store instructions are propagated to the following level of the memory hierarchy (L2 in our case). The D\$ is not-write-allocate, so that memory locations are not fetched to cache on a store miss.

ITLB and DTLB are 8-entry fully-associative, with 1KB page size. Both TLB use random replacement with the said multiply-with-carry PRNG. The TLB 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. Instructions in the execute stage check whether the value required (either through a register or through a data cache read operation) is produced by the instruction in the write-back stage.

The L2 cache size is 512KB with 16-byte lines. It is a 8-way 64K set cache implementing our random placement and random replacement policies. Table 3.1 shows the duration of *imisslat* and *dmisslat*.

3.2 Benchmark Suite

We used benchmarks from the EEMBC Autobench suite [32] for the analysis. EEMBC Autobench is a well-known benchmark suite that reflects the current real-world demand of some embedded systems. We used: a2time (A2T), aiffttr (AIFF),

Table 3.1: Duration of memory accesses (*ilat* for instructions and *dlat* for data) for each of the hit/miss combinations in each cache level

I\$	ITLB	D\$	DTLB	L2	ilat	dlat
hit	hit	-	-	hit	1	-
miss	hit	-	-	hit	10	-
miss	hit	-	-	miss	100	-
hit	miss	-	-	hit	100	-
miss	miss	-	-	hit	110	-
miss	miss	-	-	miss	200	-
-	-	hit	hit	hit	-	1
-	-	miss	hit	hit	-	10
-	-	miss	hit	miss	-	100
-	-	hit	miss	hit	-	100
-	-	miss	miss	hit	-	110
-	-	miss	miss	miss	-	200

aifrf (AIFI), aiiift (AII), cacheb (CA), canrdr (CN), iirft (IIR), puwmod (PU), rspeed (RS), tblock (TB) and ttsprk (TT) benchmarks.

Independent and identically distributed observations. The application of EVT also requires that the data are i.i.d., which is ensured by a combination of hardware with suitable randomisation properties and a suitable experimental methodology when observing the executions of the program under consideration.

We start by checking that the data are identically distributed.

- Identical distribution: We apply the two-sample Kolmogorov-Smirnov [16] test in which any two sample sets are compared. The Kolmogorov-Smirnov (KS) statistic quantifies the distance between the empirical distribution functions of two samples. The *null hypothesis* \mathcal{H}_0 is that both samples are identically distributed and the *alternative hypothesis* \mathcal{H}_1 is that samples are not identically distributed. For our experiments we use a *significance level* $\alpha = 0.05$, which is a common choice in this type of test. The outcome of the test is called the *p-value*. If the p-value is higher than α the null hypothesis cannot be rejected, which means that both samples are identically distributed.
- Independence: In order to prove that samples are independent we use the *runs test* for randomness [7]. This test is used to check whether a series of binary events can be considered to be randomly distributed, and hence independent. In this test, the null hypothesis is that the data are randomly distributed (independent) and the alternative hypothesis is that the data are not randomly distributed (not independent). A run is defined as a sequence of identical events, preceded or succeeded by different events or no events. The runs test used here applies to binomial distributions only. For example, in a sequence 0110111, we have 4 runs (0, 11, 0, 111). Our data are continuous, a cut-point must be chosen by the user so that the data are transformed into a binary sample. The binary transformation used consists of comparing each execution time with the median across all runs, so that the execution time is

replaced by 0 if lower than the median and by 1 otherwise. The expectation of the number of runs R is given by : $E(R) = 2mn/N$, where m is the number of events of type 1, and n the number of events of type 2, and N is the total sample size. The variance of the number of runs R is given by $V(R) = \frac{2mn \times (2mn - N)}{N^2 \times (N-1)}$.

Let r be the number of runs measured in the sample. It is known that when m or n tend to infinity then $Z = \frac{r - E(R)}{\sqrt{V(R)}} \rightarrow N(0, 1)$, where $N(0, 1)$ is the standard normal distribution [7]. This means that the test statistic, Z , is asymptotically normally distributed. From the value of Z , we compute the p-value with a significance level $\alpha = 0.05$. If the p-value is higher than alpha the null hypothesis cannot be rejected, which means that data are randomly distributed.

3.3 Results for Multi-Level Caches

Table 3.2: Result of the i.i.d. test

benchmark	Identical Distribution		Independence	Both test passed?
	S=50	S=100	p	
A2T	0.65	0.41	0.42	ok
AIFF	0.36	0.32	0.57	ok
AIFI	0.30	0.47	0.40	ok
AII	0.43	0.48	0.43	ok
CA	0.51	0.69	0.58	ok
CN	0.26	0.39	0.39	ok
IIR	0.44	0.35	0.53	ok
PU	0.57	0.78	0.41	ok
RS	0.23	0.26	0.30	ok
TB	0.18	0.47	0.45	ok
TT	0.60	0.42	0.54	ok

Table 3.2 shows the results of the i.i.d tests for the 8-way configuration. The second and third columns in the table show the p-value obtained by applying the KS test to the execution times obtained for a sample of 10,000 observations. From this *original sample* we create two *smaller samples* of m elements. For this experiment we use two values for m : 50 and 100. We populate the smaller samples by randomly taking elements from the original sample, which ensures that the smaller samples maintains the same statistical properties as the original one [16]. In our case, the property of interest is the distribution. Then we apply the two-sample KS test to the two smaller samples. We observe that in all cases the p-value is higher than 0.05 which indicates that data are identically distributed. Analogously, the fourth column shows the results for the independence test when using the aforementioned 10,000 observations for each program. Again, all programs pass the test. Therefore, since all benchmarks pass the i.i.d. tests, we can conclude that our architecture fulfils those properties required by PTA [13]. Other cache configurations (1-way, 32-way and 256-way) have been also tested and all of them also passed the i.i.d.

tests.

Next, we evaluate the probabilistic WCET (pWCET) estimations achieved with our PROARTIS-friendly processor when implementing our random placement random replacement caches. Table 3.3 shows the pWCET estimations we obtain for probabilities 10^{-10} , 10^{-13} , 10^{-16} when using random placement and random replacement caches (RP+RR). Results are normalised with respect to the average performance for this cache design. As shown, the pWCET is near the average and degrades slowly when the exceedance probability is decreased. For instance, the pWCET for 10^{-13} exceedance probability is 14% higher than the average performance and decreasing the exceedance probability down to 10^{-16} increases the pWCET slightly (17% higher than the average performance).

We can also compare those results with the performance obtained with an ideal processor implementing conventional LRU replacement and modulo placement (LRU+MOD ideal). As shown, conventional designs are slightly faster than RP+RR ones on average (3% higher performance). However, WCET estimation techniques built upon those designs need to know *all* addresses accessed by the program to achieve accurate WCET estimations, which is often unaffordable — if at all possible. For instance, no WCET estimation technique has been able to analyse L2 caches shared across data and instructions. Our RP+RR caches allow those shared L2 caches to be analysed. We have computed what would be the best WCET that conventional techniques could achieve in processors implementing conventional cache designs if *all* information is known but 1% of the memory addresses (LRU+MOD 1% unknown). As shown, the WCET would grow dramatically. On average, it would be more than 4X the pWCET with 10^{-16} exceedance probability for processors equipped with RP+RR caches (5.10 vs 1.17).

Table 3.3: MBPTA pWCET estimations for EEMBC benchmarks for the multi-level configuration. Results are normalised w.r.t. the RP+RR average performance

benchmark	RP+RR 10^{-10}	RP+RR 10^{-13}	RP+RR 10^{-16}	LRU+MOD ideal	LRU+MOD 1% unknown
A2T	1,36	1,43	1,50	0,96	7,80
AIFF	1,07	1,08	1,11	0,96	4,59
AIFI	1,14	1,18	1,21	0,98	6,96
AII	1,05	1,07	1,08	0,98	4,62
CA	1,06	1,07	1,09	0,97	2,53
CN	1,04	1,05	1,06	1,00	2,16
IIR	1,29	1,36	1,43	0,87	11,94
PU	1,03	1,04	1,04	0,99	2,09
RS	1,03	1,03	1,04	1,00	2,02
TB	1,21	1,25	1,31	0,96	9,04
TT	1,02	1,03	1,04	1,00	2,34
Average	1,12	1,14	1,17	0,97	5,10

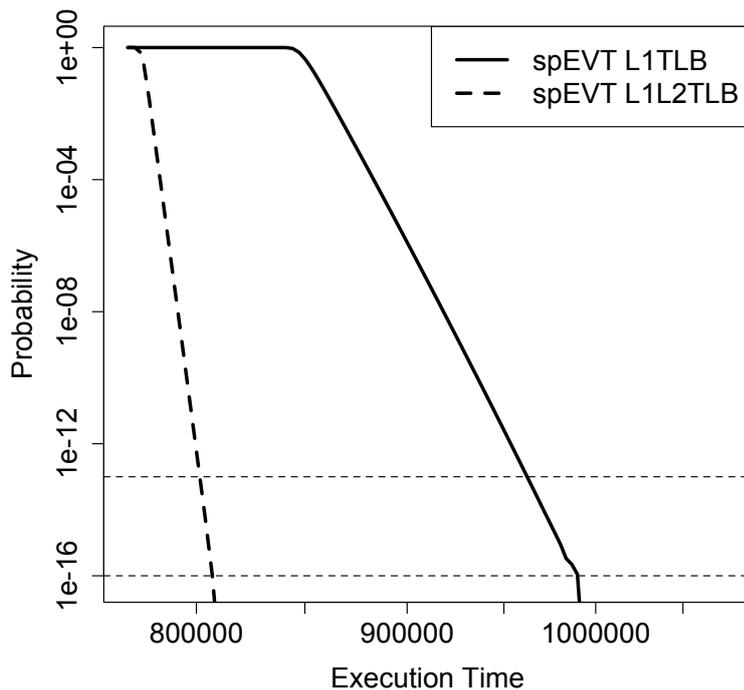


Figure 3.1: Single-path pWCET estimation for one benchmark (puwmod) with two different configurations (L1TLB and L1L2TLB)

3.4 Multi-Level vs Single-Level Caches

We have compared what would be the pWCET of a system without a L2 cache. Figure 3.1 shows the pWCET estimations we obtain with the *puwmod01* benchmark with (L1L2TLB) and without a L2 cache (L1TLB). We observe that for the range of probabilities under consideration (10^{-13} - 10^{-16}) this difference is around 20% and widens as the target exceedance probability decreases because extra cache levels introduce extra sources of random performance variation. It is well known that increasing the sources of random variation narrows distributions down because extreme cases become more unlikely. Hence, introducing a further source of variation such as the L2 cache reduces the pWCET.

Table 3.4 shows the results for all EEMBC benchmarks. In particular it shows the reduction in pWCET for each target probability under consideration when adding a L2 cache. For example the second column shows $\frac{\text{single-level pWCET}(10^{-10})}{\text{multi-level pWCET}(10^{-10})}$. For instance, *a2time* benchmark gets a 5.9X for 10^{-10} exceedance probability. This means that the pWCET for a 10^{-10} exceedance probability for multi-level cache processor is 5.9X lower (and thus, better) than for single-level cache processors. We observe that for some benchmarks the reduction in pWCET is more than 10X while the average reduction in pWCET for all benchmarks and all three target probabilities is around 3X. This result is very significant because it proves that processors can be equipped with L2 caches not only to improve average performance, but to improve pWCET, which is the key metric in CRTES.

Table 3.4: MBPTA pWCET estimations for EEMBC benchmarks the multi-level setup normalized to pWCET estimation for the single-level setup for the same target probability.

benchmark	10^{-10}	10^{-13}	10^{-16}
A2T	5.9	6.8	7.7
AIFF	2.2	2.2	2.2
AIFI	2.5	2.8	3.0
AII	2.2	2.3	2.3
CA	1.3	1.3	1.3
CN	1.3	1.3	1.4
IIR	8.1	9.1	10.2
PU	1.2	1.2	1.2
RS	1.1	1.2	1.2
TB	3.1	3.5	3.8

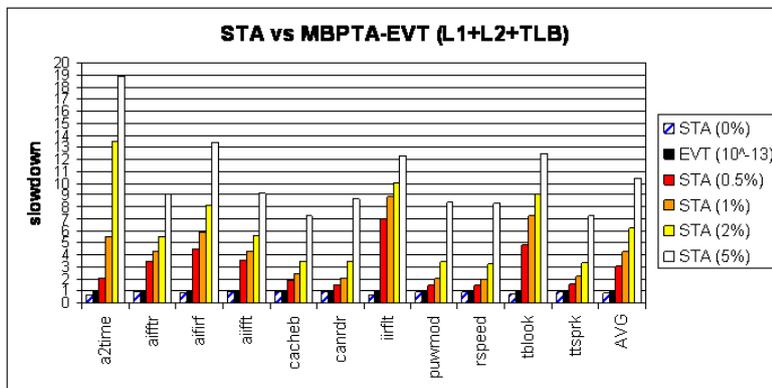


Figure 3.2: Comparison of MBPTA-EVT and STA

3.5 Comparison with Static Timing Analysis and Deterministic Architectures

In this section we extend the comparison between our PROARTIS-friendly architecture with RP+RR caches and conventional LRU+Modulo caches when using MBPTA techniques based on EVT [13] and conventional STA [37] respectively. We use multi-level caches for such a comparison and optimistic assumptions for STA since we assume that the processor and application are modelled accurately except for a fraction of the memory addresses, which are unknown. The fraction of unknown addresses varies from 0% (perfect WCET prediction) to 5% (1 of every 20 addresses is unknown). Note that since MBPTA-EVT is based on actual measurements it does not need to know any address of the program.

Figure 3.2 shows WCET results for MBPTA-EVT with 10^{-13} exceedance probability and STA with 0%, 0.5%, 1%, 2% and 5% unknown addresses for different benchmarks. Results are normalised with respect to the MBPTA-EVT pWCET. As shown before, PROARTIS-friendly architectures provide pWCET close to the performance of a processor equipped with LRU replacement modulo placement caches that knows *all* addresses of the program (15% performance difference). However,

performance degrades quickly for conventional architectures as the number of unknown addresses grows. The WCET for STA with 0.5% unknown addresses (just 1 every 200 addresses) is 3X that of PROARTIS-friendly architectures on average. Such gap quickly grows to more than 4X, 6X and 10X for 1%, 2% and 5% unknown addresses respectively. Therefore, our PROARTIS-friendly architecture together with PTA provides drastically lower WCET estimates than classical approaches.

4

Conclusions

CRTES market demands for higher performance to provide new features and increase safeness of their systems. Current processors have been shown to provide very high average performance but extremely poor WCET, which is the key metric for CRTES. Moreover, the increasing complexity of processor designs make conventional timing analysis techniques fail to provide safe and tight WCET estimates. Probabilistic timing analysis arises as the alternative to those timing analysis methods, but hardware and/or software support is required. In this work we have presented a methodology to design processors fulfilling the needs of probabilistic timing analysis while keeping design cost low. In particular, hardware designs have been proven to provide WCET estimations orders of magnitude better than conventional approaches on top of conventional processors. Moreover, we have also presented alternative software-only approaches fitting also the requirements of probabilistic timing analysis.

5

Dissemination

The work developed by Jaume Abella in this project has led to several publications. In particular, two papers have been accepted for publication, two other papers are under submission, and several other publications are still to be written but cannot be submitted yet because they are built upon material under submission and therefore, submitted papers must be accepted before publishing further results. The list of papers accepted and under submission is as follows:

- 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, D. Maxim. “*PROARTIS: Probabilistically Analysable Real-Time Systems*”, to appear in the **ACM Transactions on Embedded Computing Systems**.
- L. Cucu, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, F.J. Cazorla. “*Measurement-Based Probabilistic Timing Analysis for Multi-path Programs*”, to appear in the **24th Euromicro Conference on Real-Time Systems (ECRTS)**, 2012.
- L. Kosmidis, J. Abella, E. Quiñones, F.J. Cazorla. “*An Efficient Cache Design for Probabilistically Analysable Real-Time Systems*”, submitted to the **International Conference on Compilers Architecture and Synthesis for Embedded Systems (CASES)**, 2012.
- L. Kosmidis, L. Santinelli, M. Houston, L. Cucu, E. Quiñones, J. Abella, T. Vardanega, F.J. Cazorla. “*Towards Processor Architectures with Probabilistically Analysable Timing Behaviour*”, submitted to the **33rd IEEE Real-Time Systems Symposium (RTSS)**, 2012.

Moreover, as a result of the research performed during this project, an international patent will be filed soon. It is currently being written by the attorneys and we expect to file it around July:

- J. Abella, E. Quiñones, F.J. Cazorla. “*An Efficient Cache Design for Probabilistically Analysable Real-Time Systems Based on Random Placement*”.

Finally, an invited talk has been given at the European Space Agency:

- F.J. Cazorla, J. Abella. “*Results from PROARTIS*”, invited talk at the **European Space Agency (ESA) Workshop on Avionics Data, Control and Software Systems (ADCSS)**, Noordwijk (The Netherlands) 25-27 October 2011.

References

- [1] LLVM. <http://dragonegg.llvm.org/>.
- [2] Aeroflex Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and User's Manual*, 2011.
- [3] Aeroflex Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*, 2011.
- [4] ARM. *Cortex-R4 and Cortex-R4F Technical Reference Manual*, 2006.
- [5] ARM Ltd. *ARM920T Technical Reference Manual*, 2001.
- [6] Emery D. Berger and Benjamin G. Zorn. DieHard: Probabilistic memory safety for unsafe languages. In *In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*, pages 158–168. ACM Press, 2006.
- [7] J.V. Bradley. *Distribution-Free Statistical Tests*. Prentice-Hall, 1968.
- [8] A. Burns and D. Griffin. Predictability as an emergent behaviour. 2011.
- [9] J.M. Cargal. *Discrete Mathematics for Neophytes: Number Theory, Probability, Algorithms, and Other Stuff*. 1988.
- [10] 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. Technical Report 7869 (<http://hal.inria.fr/hal-00663329>), INRIA, 2012.
- [11] R.N. Charette. This car runs on code. In *IEEE Spectrum online*, 2009.
- [12] P. Clarke. Automotive chip content growing fast, says gartner. In <http://www.eetimes.com/electronics-news/4207377/Automotive-chip-content-growing-fast>, 2011.
- [13] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F.J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *Accepted for Publication. ECRTS.*, 2012. <http://www.loria.fr/~cuculili/ecrts2012.pdf>.
- [14] M.H. DeGroot and M.J. Schervish. *Probability and statistics*. Addison-Wesley, Reading MA., 2002.

- [15] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *the 22nd IEEE Real-Time Systems Symposium (RTSS01)*, pages 215–225, 2001.
- [16] W. Feller. *An introduction to Probability Theory and Its Applications*. 1996.
- [17] C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise wcet determination for a real-life processor. *First International Workshop on Embedded Software (EMSOFT 2001)*, 2001.
- [18] Edward Frees and Emiliano Valdez. Understanding relationships using copulas. *North American Actuarial Journal*, 2:1–25, 1998.
- [19] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading MA., 1988.
- [20] D. Griffin and A. Burns. Realism in Statistical Analysis of Worst Case Execution Times. In *10th International Workshop on Worst-Case Execution Time Analysis (WCET 2011)*, pages 44–53, 2010.
- [21] J. Hansen, S. Hissam, and G. A. Moreno. Statistical-based wcet estimation and validation. In *the 9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.
- [22] S. Huntzicker et al. Energy-delay tradeoffs in 32-bit static shifter designs. In *ICCD*, 2008.
- [23] S. Kotz and S. Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.
- [24] K. Lahiri, A. Raghunathan, and G. Lakshminarayana. LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs. In *Proceedings of the 38th annual Design Automation Conference, DAC '01*, pages 15–20, 2001.
- [25] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Wcet analysis of multi-level set-associative data caches. *9th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2009.
- [26] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, 1999.
- [27] G. Marsaglia and A. Zaman. A new class of random number generators. *Annals of Applied Probability*, 1(3):462–480, 1991.
- [28] E. Mezzetti and T. Vardanega. On the industrial fitness of wcet analysis. *11th International Workshop on Worst-Case Execution-Time Analysis*, 2011.
- [29] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems - Special issue on worst-case execution-time analysis archive*, 2000.
- [30] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. CACTI 6.0: A tool to understand large caches. *HP Tech Report HPL-2009-85*, 2009.

- [31] S.M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technical University of Munich, 2002.
- [32] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
- [33] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, November 2007.
- [34] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. *Int'l Workshop On Worst-Case Execution Time Analysis (WCET 2006)*, 2006.
- [35] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. 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), 2010.
- [36] I. Wenzel, R. Kirner, P. Puschner, and B. Rieder. Principles of timing anomalies in superscalar processors. *Proceedings of the Fifth International Conference on Quality Software*, pages 295–306, 2005.
- [37] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, G. Staschulat, and P. Stenström. The worst-case execution time problem: overview of methods and survey of tools. *Trans. on Embedded Computing Systems*, 7(3):1–53, 2008.
- [38] R. Wilhelm et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7:1–53, May 2008.
- [39] S. Zhou. An efficient simulation algorithm for cache of random replacement policy. In *the 7th IFIP international conference on Network and parallel computing (NPC2010)*, pages 144–154, 2010.