D1.2 Platform Design Guidelines for Single Core
Version 2.0

Document Information

Contract Number	249100
Project Website	www.proartis-project.eu
Contractual Deadline	m27
Dissemination Level	D1.2 Restricted*/Public[1]
Nature	D1.2 Report
Lead Authors	Eduardo Quiñones, Jaume Abella and Francisco J. Cazorla
Contributors	All members of all institutions (BSC, RAPITA, UNIPD, INRIA, AFS)
Reviewers	Tullio Vardanega (D1.2)
Keywords	Processor architecture, Random Cache Designs, Compiler, Probabilistic Timing Analysis, Single core

Notices:

The research leading to these results has received funding from the European Community's Seventh Framework Programme ([FP7/2007-2013] under grant agreement nº 249100.
©2010 PROARTIS Consortium Partners. All rights reserved.

[1]All Deliverables marked RE*/PU will be publically available within 6 months of their delivery to the EC

Change Log

Version	Description of change
v1.0	Initial Draft released to the European Commission
v2.0	Updated version in which the text accompanying reference [8] has been updated to better represent what it is said in [8] (Check the paragraph starting with 'IBM has for example found that' on page 9). In this version it has also been clarified which parts of Figure 2.1 are based on facts and which parts are intuitions of the partners (check paragraph starting with 'Our belief is' on page 12).

Contents

1 Introduction **5**

2 PROARTIS technology: Vision and Approach **7**
2.1 Background 7
2.1.1 Timing Composability 8
2.2 PROARTIS 9
2.2.1 Vision and Approach 9
2.2.2 Foundations 11

3 Towards a Processor Architecture with Probabilistically Analysable Timing behaviour **17**
3.1 Hardware Support for PTA 19
3.1.1 On PTA and its assumptions on the hardware 19
3.1.2 Mandatory Features 20
3.1.3 Providing the mandatory features 21
3.2 Probabilistically Modelling the Timing Behaviour of Processor Resources 24
3.2.1 Assigning ETP to individual resources 24
3.2.2 Composing the ETP of different resources 26
3.3 Computing the ETP of execution components using multiple time-randomised resources 27
3.4 A Taxonomy of Processor Resources 28
3.4.1 Abstract Classification 29
3.4.2 An Illustrative Example 30
3.4.3 The Case of the Branch Predictor 32
3.4.4 Hardware Resource Taxonomy 33

4 The PROARTIS Random Cache: Random Placement and Random Replacement Policies **37**
4.1 Timing Behaviour of Random Caches 38
4.2 Random Replacement 39
4.3 Random Placement 41
4.4 Generalisation of the Cache Layout Concept 43
4.5 Putting All Together: Set-Associative Caches 44
4.6 Hardware Implementation 45
4.6.1 Pseudo-Random Number Generator 45
4.6.2 Random Replacement Policy 47
4.6.3 Random Placement Policy 47

5 Compiler and Run-Time Support for Randomisation **49**
5.1 Random Cache Behaviour on Deterministic Caches 49
5.2 Software Components and Memory Objects 50
5.3 Random Location of Memory Objects 51
5.3.1 Memory Object Size 51
5.3.2 Influence of the Deterministic Placement Policy 52
5.4 Computation of ETPs at Processor Instruction Level 53
5.5 Random Software Approach Implementation: Stabilizer 54
5.5.1 Function Randomisation 54
5.5.2 Stack Randomisation 55
Acronyms and Abbreviations 57

1

Introduction

In this Deliverable we first introduce the PROARTIS vision and motivation and summarise the PROARTIS technology. In that introductory section it is shown that the use of Probabilistic Timing Analysis (PTA) requires that the timing behaviour of the platform, considered at the granularity of processor instructions, must have either no dependence at all on execution history or a dependence that can be characterised probabilistically.
In the second section, we introduce the PROARTIS platform, a Probabilistic Platform formed by the processor hardware and those low-level software components whose operation may affect the timing behaviour of processor instructions: The compiler and memory-related run-time libraries. The PROARTIS platform introduces time randomisation in both components as a way to enable the use of PTA. We define the mandatory processor features required to apply PTA and provide a detail taxonomy of how processor resources accomplish with PTA requirements. This section also presents the PROARTIS time-randomised cache, whose timing behaviour fulfills the PTA mandatory features. Finally, a software technique that enables deterministic caches to have the same required timing behaviour of the PROARTIS random cache is presented.
The evaluation results of the hardware and software techniques presented in this deliverable are shown in deliverable D3.4. This has been done in this way since we have to introduce first the PTA techniques and then show the results of the combined time-randomised hardware and PTA techniques.

2

PROARTIS technology: Vision and Approach

The market for Critical Real-Time Embedded Systems (CRTES), which includes among others the avionics and automotive sectors, is experiencing an unprecedented growth, and is expected to continue to steadily grow for the foreseeable future [9]. Let us for instance consider 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 [8] 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.

IBM has for example found that 50% of the warranty costs in cars are related to electronics and their embedded software, which cost industry billions of Euros annually [8]. In this evolving scenario, it must be acknowledged that the industrial application of current WCET analysis techniques [35], which accounts for a significant proportion of total verification and validation time and cost of system production, yields far from perfect results.

2.1 Background

Current state-of-the-art timing analysis techniques can be broadly classified into three strands: measurement based, static timing analysis and hybrid approaches using combinations of both [35].

- Measurement-based analysis techniques rely on extensive testing performed on the real system under analysis [10] using stressful, high-coverage input data,

recording the execution times observed. One common technique in industry is to measure high watermarks and add an engineering margin to make safety allowances for unknown scenarios. However, the size of a suitable 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 do not execute the code, instead they rely 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 an upper-bound on the WCET. Static approaches have limitations: 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; in that case the construction of the timing model must resort to observations.

- Hybrid measurement and static techniques use combinations of measurements and structural analysis. The approach taken by RapiTime is to take small parts of code that are measured and combined using analysis. Hybrid techniques avoid having to construct an expensive model of the hardware and allow the user to achieve a higher confidence in a WCET than simply measuring end-to-end. However such techniques require a good method of measuring on-target timing information and still rely on having an adequate level of testing.

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.

2.1.1 Timing Composability

Most static analysis techniques produce a single estimate for a single worst-case path, which makes it difficult to reason about how subsystems contribute to the execution time behaviour of the system as a whole. Measurement-based analysis suffers from the fragility of assuming that the observation of the system in the test environment will match the behaviour after deployment, and that an engineering safety margin is sufficient to protect against unexpected increases in execution time. Hybrid analysis strikes a balance between the other methods, and improves a great deal on the accuracy of measurement-based analysis and the complexity of static analysis.

All of these approaches have a common problem: that of a lack of composability. The results obtained for components of a system are either specific to that system, and cannot be reused when the components are combined in a different configuration, or – if they are – they incur undue pessimism. This makes some engineering goals, such as incremental qualification, extremely difficult to achieve.

2.2 PROARTIS

2.2.1 Vision and Approach

PROARTIS 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 (Symmetric MultiProcessing) SMP architectures, the software which may be running on another core sharing the same system bus and RAM. A consequence of those dependences 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 this deliverable and deliverable D3.4, we 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 new probabilistic analysis techniques. 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.

Applying time-randomisation techniques has inevitable consequences for the average-case execution time of a program. Figure 2.1 illustrates the PROARTIS view of execution time [1]. The execution time shift between deterministic and time-randomised architectures is marked (a): In general we expect the time-randomised profile to

[1] In deliverable D3.4 we quantify the intuition-based results shown in Figure 2.1.

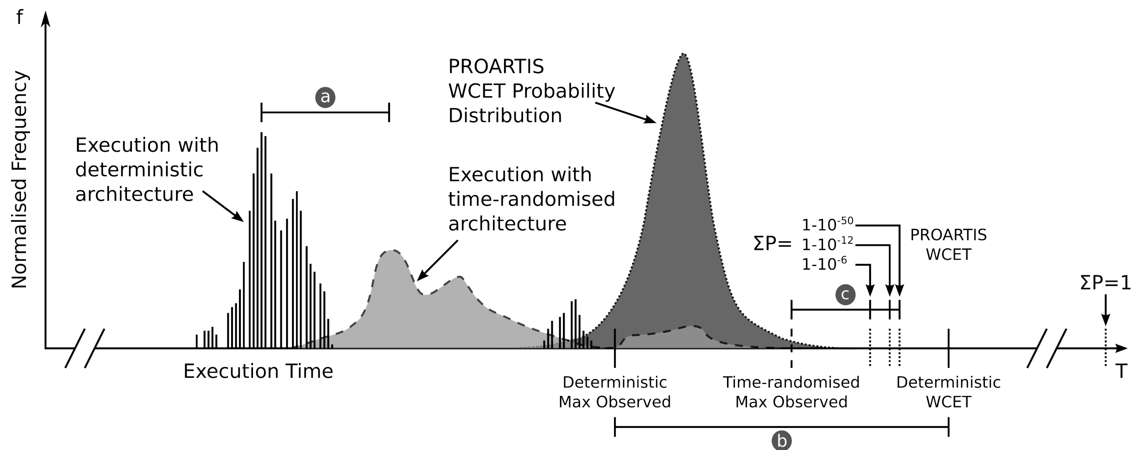


Figure 2.1: Execution time distributions for conventional deterministic architectures and a proposed time-randomised architecture, superimposed with the PROARTIS worst-case probability distribution

shift to the right (to increase) in relation to the deterministic profile, and to spread across a larger range of execution times. However the resulting distribution becomes more predictable: by decoupling timing events (e.g., cache accesses), they compose into a smooth curve, with a long tail describing execution times which are increasingly unlikely. Dependences between events in deterministic architectures can have an abrupt impact on execution time, producing discontinuities in the possible execution times which are difficult to model with a parametric distribution.

The absolute maximum execution time produced by the PROARTIS analysis will be many times greater than a conventional WCET, as it will correspond with the case where all instructions take the longest possible time to execute (e.g., every cache access is a miss [30]). We expect instead to gain by tightening the gap between observed execution times and worst-case execution bounds using a probabilistic confidence limit. The result of static analysis of deterministic architectures produces a degree of pessimism, where unknown states must be considered to have their worst consequences on the timing of the system. The 'true WCET' lies somewhere in the range marked (b) in Figure 2.1, between the maximum observed execution time and the WCET bound produced by analysis.

In PROARTIS the consequences of these unknown states can be considered probabilistically, which enables us to reason about the WCET probabilistically. Techniques from Extreme Value Theory (EVT) are used to construct a worst-case probability distribution. We define worst-case bounds with stated confidence levels, which can be chosen to match the degree of uncertainty present in the rest of the system under analysed. Our belief is that probabilistic WCET (pWCET) bounds are considerably more resilient to the lack of execution state information about the system being analysed than classic static timing analysis (STA). This resilience translates in STA incurring increasingly more pessimistic WCET bounds as less execution state information is available. This degradation occurs because STA *must* assume the worst case when no better hypotheses can be sustained, and the resource-level worst case for deterministic processor architectures can be very high.

In the reporting period we have collected some initial experimental evidence to support our hypothesis: the results are presented and discussed in Section 4.6 of D3.4. This initial evidence corresponds exactly to the claim represented by area

(c) in Figure 2.1.
WCET estimates are then computed by considering the execution time at which the cumulative probability (ΣP) exceeds the required level of confidence. These confidence levels are expressed in Figure 2.1 in terms of the probability of exceeding the WCET threshold *in any given run*, however this figure should be adjusted based on arrival frequency to determine the probability per hour, or per year as necessary.

2.2.2 Foundations

By modelling the timing behaviour of system components as random variables, a sound mathematical basis can be introduced upon which statistical methods for execution time analysis may be applied. This means that WCET estimates can be given with a confidence value, rather than being an absolute, but pessimistic guarantee.

Platform Requirements The PROARTIS method of Probabilistic Timing Analysis (PTA) rests on two *fundamental premises*, which must be warranted by the system of interest, inclusive of its processor hardware, the Operating System and the application software:

1. The timing behaviour of the processor hardware, as observed at the granularity of processor instructions, must have either no dependence at all on execution history or a dependence that can be characterised probabilistically.

2. The timing behaviour of the application software, as observed at the granularity of procedures (subprograms) must be time composable, this meaning that the bound determined for each individual procedure must stay valid (i.e., true upperbounds) - and possibly but not necessarily as tight - in the face of its composition with other procedures and ultimately after integration in the full system.

The fulfilment of Premise 1 is specifically discussed in Section 3 and a publication produced by the PROARTIS team is being submitted to a top-tier conference at the time of this writing. The essence of the PROARTIS strategy to achieve the property evoked by Premise 1 is that the hardware resources used by processor instructions should be divided in two categories: those whose response time jitters by an amount that can be bounded from above (i.e., by fixing it to the worst-case) without incurring excessive pessimism; and those whose response time jitter cannot be bounded from above without incurring unacceptable pessimism and therefore must be redesigned to randomise their timing behaviour so that it is subject to probabilistic laws that have no (or probabilistically boundable) dependence on the history of execution. A similar principle is followed to remove or bound from above software-introduced variability, both at application and operating system level. Overall, the observations taken to feed the analysis method capture all the variability that can appear at run time, ensuring that the pWCET estimations obtained from runs done during the testing phase bounds the execution time of the system during deployment time.
A typifying example of a processor resource with high jitter is the cache memory (regardless of whether for data or instructions). In Sections 3 and 4, in addition

to presenting a taxonomy of all processors resources in the regard of their response time jitter, we show that the cache memory can be redesigned using a high-quality long-period pseudo-random number generator to drive its response time. We also show that the performance of such a time-randomised cache (which uses random placement and random eviction on access or miss): can be economically implemented; incurs an acceptable performance degradation; has a residual dependence on execution history in the eviction policy, which, while depending on the considered execution path, retains no dependence across paths so that it can be used in a probabilistic timing analysis that studies multiple paths. Finally, we present a randomise compiler and run-time system, named *Stabilizer*, that allows caches implementing a deterministic modulo placement policy to behave as having the PROARTIS random placement policy and so fulfilling Premise 1.
To address Premise 2, instead, deliverable D2.2 presents a novel approach to the design and implementation of a time-composable operating system, whose service calls have a tightly upperbounded response time and whose execution has no negative effect on the timing behaviour of the caller after return. Interestingly, this notion of time composability is equally beneficial for classic timing analysis for standard platforms, but its fuller benefit manifests itself for PROARTIS systems where the entire system construction seeks independence of timing behaviour at all levels of abstraction.

Timing Analysis PROARTIS pursues three approaches to PTA:

- Static PTA (SPTA), the timing events that correspond to the random variables of interest are determined statically from a model of the processor and the software. Static PTA is performed by calculating the convolution of the discrete probability distributions which describe the execution time for each processor instruction; from this we obtain a probability distribution, or Execution Time Profile (ETP), which represents the timing behaviour of the entire sequence of instructions. Static PTA can be applied to single, linear paths. The results obtained from this step of application can then be combined into a pessimistic envelope profile which covers the worst-case path(s), using techniques in use with industrial quality tools like RapiTime.

- Measurement-Based PTA (MBPTA), in which, end-to-end measurement runs of the program under study (again, linear paths of program traversal) are made on the PROARTIS hardware. The resulting information is used to determine the timing profile - as an execution time frequency distribution - of individual elements (paths) of the system, and then of the system as a whole. This procedure critically depends on the availability of input data that warrant sufficient path coverage: whereas in classic STA full path coverage must absolutely be achieved, a measurement-based PTA only needs a probabilistically quantified measure of coverage, to be applied to the computed timing profile, which can be obtained from recording all (source and object) paths traversed during measurement runs and relating this information to the observed execution time values. In PROARTIS, MBPTA provides pWCET estimate for observed paths during the end-to-end measurement runs.

- Hybrid PTA (HyPTA) follows the same principle of operation used by RapiTime,

that is a combination of static structural analysis and measurements from testing. HyPTA uses the measured times for small segments of code, which typically correspond to source code blocks. For each block MBPTA is performed, with which HyPTA builds a probabilistic "envelope" that produces the pWCET estimate of the entire program. That is the profiles of the observed paths are combined using the structure of the code to obtain an overall WCET estimate for the entire program, including synthetic paths which may not have been tested.

The PROARTIS method of measurement-based probabilistic timing analysis (MBPTA for short) builds on the provable fulfilment of Premises 1 and 2 discussed above and of their related corollaries. The MBPTA method and its theoretical foundation are presented in a publication to appear in the proceedings of ECRTS 2012, a world-wide acclaimed top-tier real-time systems conference [11].

The MBPTA approach to worst-case execution time analysis proposed by PROARTIS builds on the application of Extreme Value Theory (EVT) on the postulate that worst-case execution times can be regarded as rare events. The input values to be analysed with EVT techniques are execution times obtained from measurement runs of the program units of interest. The use of EVT for our purposes requires (1) that those values can be regarded as independent and identically distributed random variables [2]. The two premises discussed at the start of this section serve to warrant this hypothesis. And (2) that the pWCET estimations obtained from execution time observations during testing are guaranteed to bound from above the execution times that can be incurred during system operation. PROARTIS achieves this important property by ensuring that all sources of execution time variability are either upperbounded so that those sources have the same behaviour during testing and operation; or time-randomised which enables us to use observations to derive conclusions about unobserved execution times.

The application of EVT further demands the choice of a specific EVT-conforming distribution with which the events of interest are to be studied. Of the three EVT-conforming distributions that are known to the state of the art, the PROARTIS MBPTA method has chosen the Gumbel distribution, which is shown to best accommodate the problem of modelling the tail end of the execution time distribution, hence its worst-case behaviour.

Each of the individual distribution functions is characterised by a specific choice of three discriminating parameters known as: the shape; the scale; and the location. The choice of a given distribution, Gumbel's for PROARTIS MBPTA, requires the determination of those three parameters and the demonstration that they provably fit the chosen distribution characteristics. On account of all of these prerequisites, the PROARTIS MBPTA method proceeds as follows:

(a) A number of execution-time observations (measurement runs) have to be made for each and every program unit of interest. The cited publication [11] proposes an empirical method, which it experimentally demonstrates to determine that a sufficient number of observations has been made.

[2]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. A more detailed explanation of both concepts is given in deliverable D3.4

(b) The principles of EVT theory are applied to the product of step (a) by grouping the observations in sets, and sampling the maxima from each set. This method, known as block maxima, serves the purpose of fitting the tail end of the execution time distribution, which captures the worst-case timing behaviour. Various criteria exist to determine the most convenient size of such sets; an obvious trade-off occurs here between accuracy and cost.

(c) An iterative process is used to arrive at the point where a best-fit Gumbel distribution is found for the data selected from the observations. This process may require additional observation data to be supplied and different selection (refinement) of the distribution-characterising parameters (shape, scale and location).

Step (a) of the above procedure stays the same regardless of whether the program unit (procedure) is single-path or else includes multiple paths; of course, multi-path program units are to be regarded as the norm and not the exception. When step (a) is applied to a multi-path program unit, care must be taken to ensure that the single, compound distribution obtained from the observation of measurement runs traversing multiple paths still exhibits the required property of independence and identical distribution. This may be achieved either by selecting inputs randomly when running the measurement runs and grouping them sequentially (step (a)), or by testing all inputs and selecting results randomly, without replacement, when performing grouping (step (b)). We assume that there is a direct and traceable correspondence between the input (data and state) to a measurement run and the path taken by the execution. Obviously, the question arises as to the validity of the results obtained from this method in the face of the path coverage attained by the measurement runs. Two comments are in order in this regard: (1) We assume and in fact even require that the measurement runs are performed as part of functional testing, so that the application of the above-described MBPTA method attains no less path coverage than that claimed by functional test for the system. The fact that, in force of Premise 2, a PROARTIS-compliant system is time composable by construction, makes this process only marginally most onerous for the user since all that the PROARTIS method requires additionally is the achievement of a minimum number of observations per path, which may be larger than that required by functional testing alone. (2) The results obtained by the method are only valid for the path coverage that is actually attained, and that can be recorded with modest enhancement of the test environment. No claim can be made for the unobserved paths. Yet, methods exist to determine unobserved paths of interest, i.e., those that could provide longer execution time than that captured by the observations (which could be done using the current functionality of RapiTime) or to combine, pessimistically, the observations made for leaf program units along a path-aware tree-based structure of the control flow graph of coarser-grained program units. The latter option is the approach used by HyPTA, which applies MBPTA for every block and builds a probabilistic envelope from it for the entire program.

An important feature for PROARTIS is that the tail-end of execution time distributions obtained by the analysis are composable; that is, that a system level analysis may be performed by combining pWCET results from component-level

analysis. This can be achieved by ensuring that the estimated distributions are independent of other effects in the system, or at least designing the system in such a way that the maximum reciprocal effects can be bounded from above and accounted for during system verification.

Independence is a core consideration of PROARTIS, but it is important to recognise the circumstances in which it must apply. Clearly, not all dependence can be eliminated from the system, as this would defeat many of the performance improving features we are attempting to enable. It appears to be sufficient that we attenuate (or ideally even eliminate) the effect of execution timing history on timing behaviour. That is, we would like to ensure that if the time one event takes to execute, regardless of whether short or long, does *not* affect the probability distribution of the execution time of subsequent events. Warranting this property enables the relevant timing distributions to be combined using a convolution operation, both at the instruction and component levels. Otherwise, another combination strategy must be used which takes the potential dependence into account at the cost of incurring larger pessimism.

The essence of the PROARTIS strategy to achieve time composability as defined above works as follows:

- The fulfilment of Premise 1 ensures that the timing behaviour of the processor does not carry dependence on execution history across program runs. This is true by definition for all processor resources whose response time has been fixed to the worst case. This can also be made to hold for time randomised processor resources in one of two ways: (1) the state of the time randomised processor resource is considered to be, for static analysis, or put in, for measurement-based analysis, the worst case before the program unit of interest is analysed; (2) a probabilistic characterisation is derived for the perturbation of the state of the time randomised resource, from the perspective of the program unit under analysis, as resulting from all executions occurring in between two subsequent runs of the said program unit. On top of this, run-to-completion semantics is assumed (and it is actually required) for all such program units so that the interference effect resulting from pre-emption do not need to be considered in the analysis.

- With the above hypothesis in force, no perturbation can occur on the timing behaviour of a program unit from other program units at the application level; moreover no further state retention may occur at the level of the processor hardware that can cause the execution time of the program unit of interest to incur variations that exceed the upper bound determined by PROARTIS analysis. It therefore follows that the sole other source of perturbation can be the Operating System in that the response time of calls made to it by the program unit of interest (whether implicitly, at the start and the end of relevant execution, or explicitly, as part of the unit code at the application level) may vary as a result of previous invocations by the same or other program unit. This disturbance, which may potentially break time composability is avoided by construction in PROARTIS by a novel design of the Operating System. The approach to achieve this goal is discussed in deliverable report D2.2.

Results The results obtained so far by the PROARTIS team on the PROARTIS execution platform confirm the soundness of the PROARTIS approach. Our results show that, unlike standard deterministic platforms, the PROARTIS execution platform (hardware and software) meets the PTA requirements. This PTA-friendly architecture has an average performance loss with respect to a deterministic architecture of less than 20% for the configurations used for the experiments presented in this report. The SPTA and MBPTA techniques have been shown to provide effective pWCET estimations. The latter covers both single-path and multipath programs. For the setup we used, the WCET estimations we obtain are comparable to those achieved by static timing analysis. When some of the information that STA requires is missing, e.g. the address of some accesses to memory, the pWCET estimations obtained with our PTA techniques are several times (from 2x to 6x) better than those achieved by STA. Moreover, the novel design and implementation of a time-composable operating system further aids in achieving and preserving time composability at the application level.
HyPTA is applied to a flight guidance system for a missile applied on a processor simulator and the current results are provided in deliverable D3.4. At this stage we are able to show that the results provided by RapiTime at the level of basic blocks are sufficient to build a probabilistic envelope. This conclusion is based on the fulfilment of all hypotheses of MBPTA that is applied to the level of basic blocks.

Overall, we have developed the theoretical basis and the support techniques to effectively enable and apply Probabilistic Timing Analysis for single-core systems. The results include a clear specification of the platform (hardware/compiler and software) requirements to enable the use of PTA and the actual implementation of PTA techniques per se. We have developed the PROARTIS execution stack technology and tool-chain needed to verify the whole range of PROARTIS technology for timing analysis, hardware simulation, compilation, operating system and application software to be thoroughly evaluated. We have also evaluated the PROARTIS solutions against: EEMBC and Mälardalen benchmarks; PROARTIS specifically-designed benchmarks (PSB); and, more importantly, the AFS case studies and Rapita's Missile Guidance System demonstrator.

3

Towards a Processor Architecture with Probabilistically Analysable Timing behaviour

Probabilistic Timing Analysis (PTA) [7] allows accurate modelling of the execution time of the execution component of interest, considered at a given 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 occurrence[1].
This probabilistic timing behaviour can be described by an Execution Time Profile (ETP for short), which, for an execution component R_i, is expressed by a pair of vectors $ETP(R_i) = \{\vec{t_i}, \vec{p_i}\}$, where $\vec{t_i}= \{t_i^1, t_i^2, ..., t_i^{N_i},\}$ captures all possible execution times of R_i, and $\vec{p_i}= \{p_i^1, p_i^2, ..., p_i^{N_i},\}$ the corresponding probabilities of occurrence, with $\sum_{j=1}^{N_i} p_i^j = 1$. In other words, the timing 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.
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. Through the hardware approach proposed in PROARTIS this is achieved by ensuring that the i.i.d. property holds at the level of processor instruction. Moreover, PROARTIS also evaluates software techniques to force deterministic hardware, i.e. cache modulo placement policy, to behave as the

[1]True probability is different from frequency. We refer the reader to Section 3.2.1 where we elaborate on this important observation.

proposed PROARTIS hardware.
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, focusing on the hardware. Other authors who studied PTA considered other levels of abstraction: for example, Petters looked at basic blocks in [29]; Burns and Griffin at time bands in [5]. Section 5 focuses on the software techniques.

Approach The PROARTIS platform has been designed so that every individual 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 convolved[2] 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 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 provides, 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 techniques[3]. 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

[2]In 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.

[3]Of course, a well-structured, timing-aware program makes the quest for flow facts and code-level analysis in general much simpler, and PTA in any flavour is not exempt from that because it will certainly require path-related knowledge to be acquired.

high-performance resources, such as caches and TLB, and yet is provably amenable to PTA. (iii) We show that our example processor passes both independence and identical distribution tests.

3.1 Hardware Support for PTA

3.1.1 On PTA and its assumptions on the hardware

In Section 2 and in [7] we showed that three PTA variants exist: static PTA (SPTA), measurement-based PTA (MBPTA) and hybrid PTA (HyPTA). 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. HyPTA derives probabilities from the combination (as probabilistic envelope) of the results of the application of MBPTA at the level of basic blocks. Therefore HyPTA has the same requirements as MBPTA on the hardware; in this section we present then only the requirements of MBPTA and SPTA.
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 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 [21] (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 [7, 12, 17, 18].

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 to 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.

3.1.2 Mandatory Features

We argue that the processor architecture must possess certain 'mandatory features', denoted $\{M_1, \cdots, M_n\}$, 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 3.1 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.

[$\mathbf{M}_1$] 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 $\mathbf{M}_2$ below.

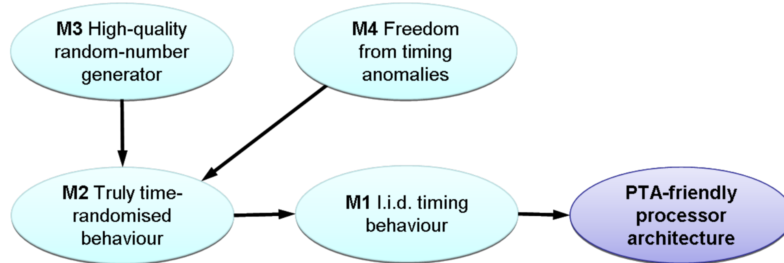


Figure 3.1: The mandatory features to be exhibited by PTA-friendly architectures and the precedence relation among them

[$\mathbf{M}_2$] The timing behaviour of the resources employed at the level of granularity set for mandatory feature $\mathbf{M}_1$ must either be truly randomised or allow bounding from above.

In this section 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 [22] (in Section 4 we propose a time-ransomised cache that accomplishes with PTA requirements). 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.

[$\mathbf{M}_3$] The hardware must be equipped with a high-quality random number generator that enables processor resources to exhibit time-randomised behaviour at low cost.

[$\mathbf{M}_4$] The hardware must be provably free of timing anomalies [24, 34], 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 3.1, from the most basic features, $\mathbf{M}_3$ and $\mathbf{M}_4$, culminating with $\mathbf{M}_1$. 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.

3.1.3 Providing the mandatory features

$\mathbf{M}_3$: 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 [33]. The multiply-with-carry PRNG is known to provide the above properties [25] and thus it is our choice. Section 4.6 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 [24, 32, 34]. 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 [34] 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 [32] 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.

$\mathbf{M}_1$: 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 $\mathbf{M}_4$, 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 [4] that measures whether binary events are biased, or else they 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 [13], 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 3.2.2 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, 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.

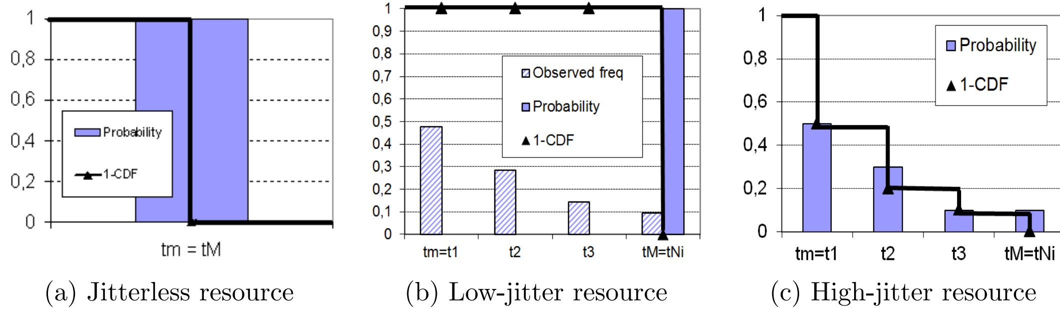
3.2 Probabilistically Modelling the Timing Behaviour of Processor Resources

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 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.

3.2.1 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. Note that the hit/miss probability is different from the frequency of events. This



(a) Jitterless resource (b) Low-jitter resource (c) High-jitter resource

Figure 3.2: Probabilistic timing behaviour of a single instruction for each type of resource

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 3.2 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 $i = 2, .., N_i - 1$ 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 3.2(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 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 3.2(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 3.2(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 3.2(c) depicts the probabilistic timing behaviour of a high-jitter resource,

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

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. In Section 3.2.2 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.

3.2.2 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 $p_c(n) = \vec{p_1} \otimes \vec{p_2}\ (n) = \sum_{k=-\infty}^{\infty} p_1(k) \times p_2(n-k)$ and $\vec{t_c}$ results from the summation of the contributing latencies, which, for all k such that $p_1(k) \times p_2(n-k) \neq 0$ is defined as $t_c(n) = t_1(k) + t_2(n-k)$.
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., [15]) which assume full dependence between probability distribution functions, but we do not consider them here.
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 below the level at which the observations are made and the i.i.d. property of those ETP is preserved across all of the operation that occurs in between the corresponding two levels of abstraction: 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 3.2(a), with the transition from probability 1 to 0 occurring at time instant $t'_m = t^1_m + t^2_m = t'_M = t^1_M + t^2_M$. 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

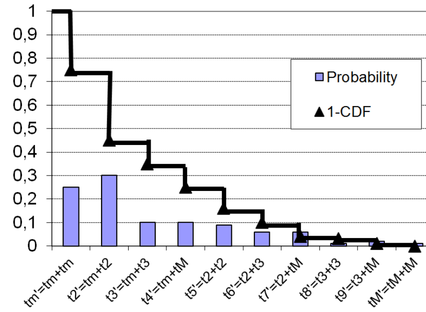


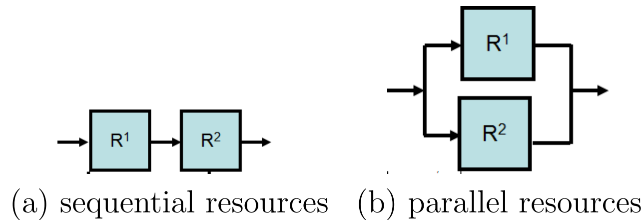
Figure 3.3: Composition of the ETP of two requests using time-randomised resources

$t'_M = t^1_M + t^2_M$.
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 3.3 shows the result of convolving two requests using the resource whose ETP is shown in Figure 3.2(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.

3.3 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 3.4(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 3.4(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



(a) sequential resources (b) parallel resources

Figure 3.4: Different resource arrangements

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 $ETP_{R_c} = \{\{2, 3, 4\}, \{0.12, 0.28, 0.6\}\}$.
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 $\mathbf{M}_2$. 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 3.1.3 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.

3.4 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 time randomisation; by enforcing the worst-case jitter by hardware or by the way the observation 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 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.

3.4.1 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 solely by the current event under consideration (no history dependence) or by the combination of previous events and the current one (history dependence); and (ii) whether the jitter is deterministic, probabilistic or simply propagated regardless of its source.
Let us first illustrate the general cases that result from the above taxonomy, with some examples before classifying processor resources in our concrete case:

- 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.

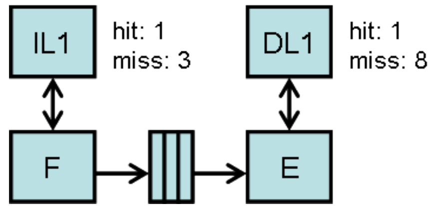


Figure 3.5: Simple processor design used in the example

- 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 observe are fixed given our assumption that the user determines the relevant paths that will be covered by the measurement runs. Note that nothing can be said for the paths that have not been observed. 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.

3.4.2 An Illustrative Example

Let us clarify why HD+JP resources do not violate the i.i.d. property required for PTA in a 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 3.5). 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 in 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, $i1$ always hits in IL1 and has a 0.9 hit probability in DL1. The remaining instructions do not access DL1.
In this code example, $i1$ may introduce some delay in the execution of the program when accessing DL1: if it misses it will cause a 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.

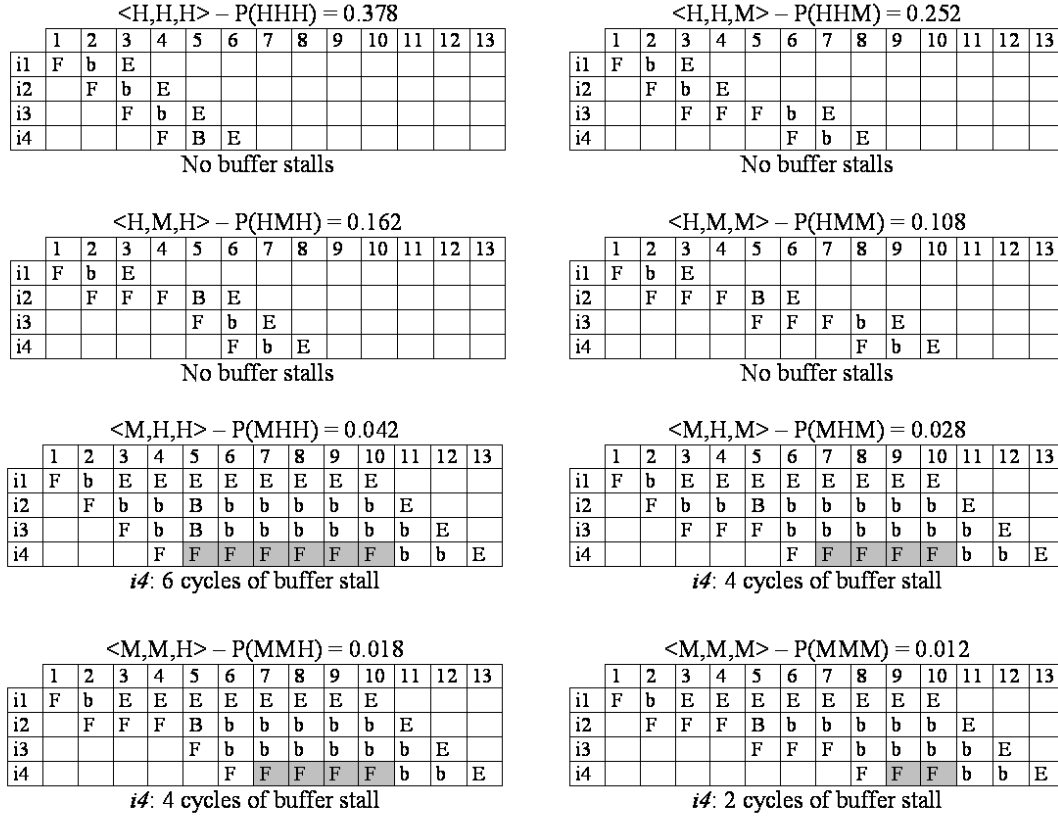


Figure 3.6: Potential chronograms based on the outcome of the different cache accesses

```
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 3.6). In particular, we use the vector <DL1i1, IL1i2, IL1i3> 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 *i*1 and *i*4 have IL1 hit probability of 100% so for this reason IL1i1 and IL1i4 do not appear in the vector.
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 <M,H,M> and <M,M,H> 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 determined by the user. Therefore, HD+JP resources (the buffer in our case) cannot create further

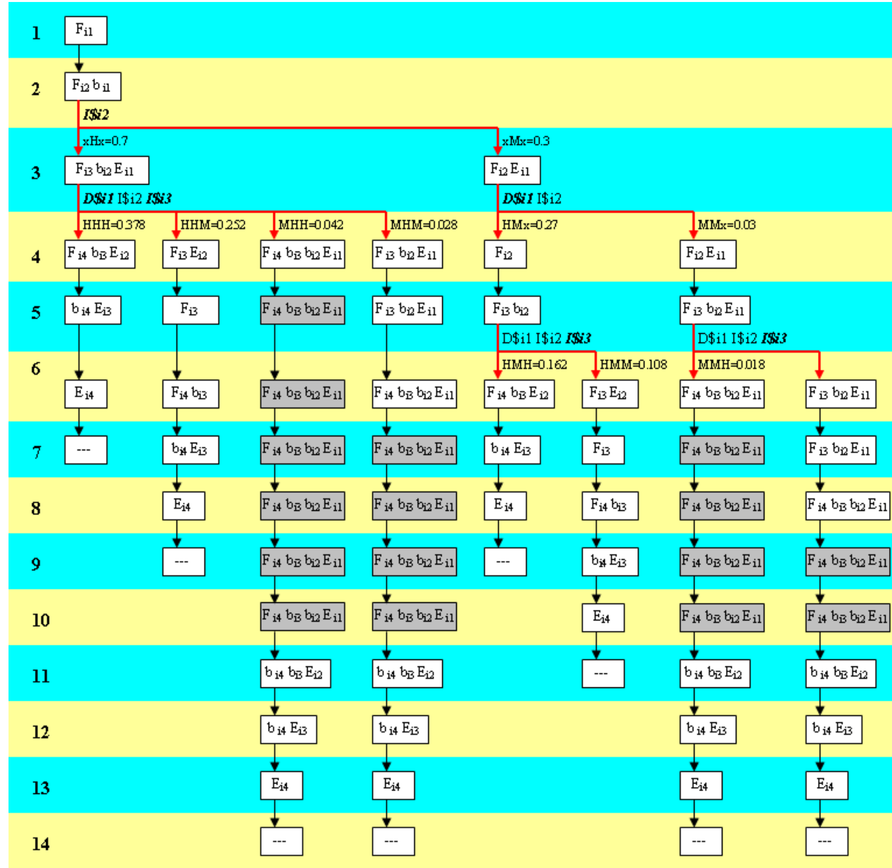


Figure 3.7: Processor Stage Graph

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 3.7 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.

3.4.3 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 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.

3.4.4 Hardware Resource Taxonomy

We consider a pipelined processor with in-order fetch, dispatch and retirement of instructions (see Figure Figure 3.8) [5]. We consider a pipelined processor with in-order fetch, dispatch and retirement of instructions. Fetch and execution stages are equipped with first 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.

Table 3.1 lists 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.

[5] Our processor pipeline is fully described in the result section of D3.4. Here we provide an overview of its main components

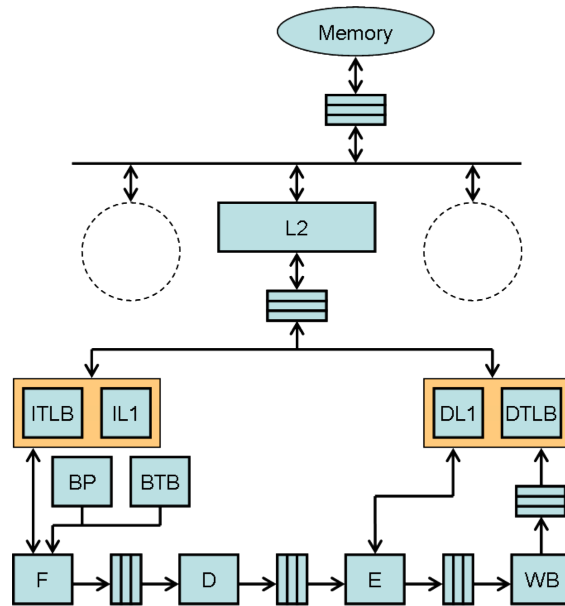


Figure 3.8: Processor components considered within PROARTIS

Table 3.1: 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 execution that preceded that of the current unit of analysis (program, in classic terminology) have completed, there cannot be any residual data-dependent effect. However, in a pipelined processor, instructions of different units of analysis executed consecutively can coexist in the pipeline and some stalls can occur if the resource is still in use. If those different units of analysis are analysed separately their WCET estimates consider the finalisation of the last previous instruction, as if those units of analysis did not overlap their execution at all. Therefore, stalls due to interleaved execution of multiple units of analysis are upperbounded by using their individual WCET estimates in a processor free of timing anomalies.				

4

The PROARTIS Random Cache: Random Placement and Random Replacement Policies

This section evaluates different cache designs that meet the PTA requirements. Caches are of prominent importance to boosting average performance since they hide the long latency of memory operations, and it is one of the main processor resources of interest to WP1 in PROARTIS.

Caches typify the problems that PROARTIS tries to overcome. Providing tight WCET estimates 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 [14] [27] [23] [31]. 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 performed by the program under study. Moreover, due to the increasing complexity in the software design, the effort to acquire knowledge about execution history of a program significantly increases [26]. When the required knowledge is not available, 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 are not achieved with current processors whose operation cannot be modelled with true probabilities, consequently breaking the fundamental assumptions required by PTA. 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 that fulfil the PTA properties and allows modelling the timing behaviour of memory operations with true 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.

4.1 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 lines [1] (cells) arranged in S sets

[1] To reduce the traffic overhead between the main memory and the cache consecutive memory

(conceptually rows) and W ways (conceptually columns). The set in which a piece of data is placed in 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 behaviour 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.
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 4.6.

4.2 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 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 remainder 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 4.1 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

objects are grouped into blocks called *cache lines*

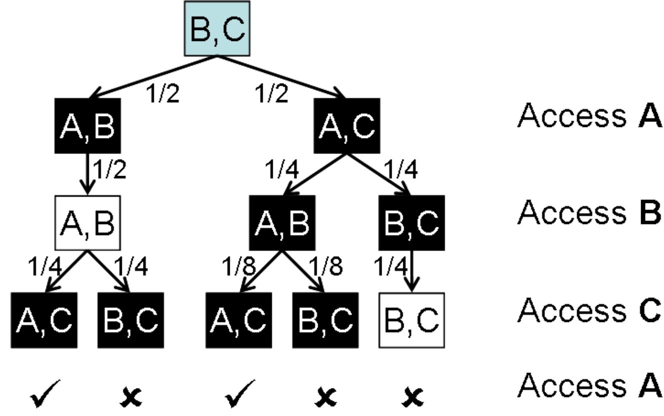


Figure 4.1: Probability tree of the sequence A, B, C, A

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 made in [36], 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 $A, B_1, B_2, ...B_x, A$, the probability of the second occurrence of A to be a hit, and so of A to survive, can be formulated as follows, where $P\{\overline{B_i evictA}\}$ is the probability that B_i does not evict A:

$$P_{hit_A} = P\{\overline{B_1 evictA}\} \cdot ... \cdot P\{\overline{B_x evictA}\} \tag{4.1}$$

In which each of the $P\{\overline{B_i evictA}\}$ 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 $(W-1)/W$.

$$P\{\overline{B_i evictA}\} = P_{hit_{B_i}} + P_{miss_{B_i}} \cdot \frac{W-1}{W} \tag{4.2}$$

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

$$ETP(A) = \{t_{hit}, t_{miss}\}\{P_{hit_A}, 1 - P_{hit_A}\} \tag{4.3}$$

Therefore, the use of random replacement policy allows to derive an ETP for each memory operation, thus enabling PTA. In the results presented in deliverable D3.4 we show that (1) the observed execution times of a FA cache with RR match those of the convolution of the ETPs of each memory request derived as shown above; and (2) the resulting execution times can be modeled with i.i.d variables, for which we use the proper statistical tests: Kolmogorov-Smirnov [13] for identical distribution and the Wald-Wolfowitz [4] for independence.

4.3 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 by 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 4.6. 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 that address to a given cache set stays the same, i.e. $\frac{1}{S}$. The hardware implementation of our random placement policy is detailed in Section 4.6.
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 mapping resulting from 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 4.1 identifies all possible cache layouts of a program consisting of $u = 3$ memory objects mapping into different cache lines (A, B, C, A). The subscript in each address in the first column indicates the cache set on which each address is mapped, 0 or 1 in this example.
With random placement we can derive the probability of each cache layout to occur. The column labelled as P_{layout} in Table 4.1 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

Table 4.1: Possible cache layouts for the different accesses of the sequence *A, B, C* in a idealised 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)	$(\frac{1}{2})^3$

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 4.1, 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 uth *moment of a probability distribution of random permutations* [16]:

$$E(X^u) = \sum_{j=1}^{S} S(u,j) \tag{4.4}$$

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 [6] with parameters u and j. In other words, the uth 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 4.1.
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 4.1, 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 $A, B_1, B_2, ...B_q, A$ 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 of 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:

$$P_{hit_A} = \frac{(S-1)^q \cdot S}{S^{q+1}} = \left(\frac{S-1}{S}\right)^q \quad (4.5)$$

The reuse distance of A, defined as the number of unique addresses (q) between two 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, B, A ($q = 1$).
To sum up, random placement guarantees by design that an ETP per processor instruction (memory request) exist. Although the resultant ETP is not independent of previous execution history, its dependence can be characterised probabilistically through q, fulfilling the platform requirements introduced in Section 2.2.2.

$$ETP(A) = \{t_{hit}, t_{miss}\}\{P_{hit_A}, 1 - P_{hit_A}\} \quad (4.6)$$

As in the case of random replacement, in Section 4.2 we show that for a given program (1) the observed execution times of a direct-mapped (DM) cache with RP correspond to the convolution of the ETPs of each memory request derived from equation 4.5; and (2) the resulting execution times can be modeled with i.i.d variables.

4.4 Generalisation 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 mth moment:

$$E(X^m) = \sum_{j=1}^{W} S(m, j) \quad (4.7)$$

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 mth 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.

4.5 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 meet the PTA requirements.
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 of[2] $1/W$:

$$P_{hit_A} = \left(\frac{S-1}{S}\right)^q \cdot (P\{\overline{B_1 evictA}\} \cdot ... \cdot P\{\overline{B_x evictA}\}) \tag{4.8}$$

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 uth and m^{th} moments of a probability distribution of random permutations:

$$E(X^m) \cdot E(X^u) = \sum_{j=1}^{W} S(m,j) \cdot \sum_{j=1}^{S} S(u,j) \tag{4.9}$$

$E(X^u)$ and $E(X^m)$ are the number of cache conflict layouts given by the random placement policy and the random replacement policy respectively. Table 4.2 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

[2]Note that this equation is not valid for direct-mapped and fully-associative caches, whose analytical models are described in equations 4.2 and 4.5 respectively.

Table 4.2: 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.

4.6 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.

4.6.1 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) [25] PRNG satisfies these requirements. The MWC PRNG produces random numbers based on the following set of equations:

$$seed_z = 36969 \cdot (seed_z \& 65535) + (seed_z \gg 16) \tag{4.10}$$

$$seed_w = 18000 \cdot (seed_w \& 65535) + (seed_w \gg 16) \tag{4.11}$$

$$RII = (seed_z \ll 16) + (seed_w \& 65535) \tag{4.12}$$

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 (4.10) and (4.11)). Recommended initial values for $seed_z$ and $seed_w$ are 362436069 and 521288629 respectively [25].
We provide an efficient implementation of MWC that we call PTA-MWC (Figure 4.2). 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.

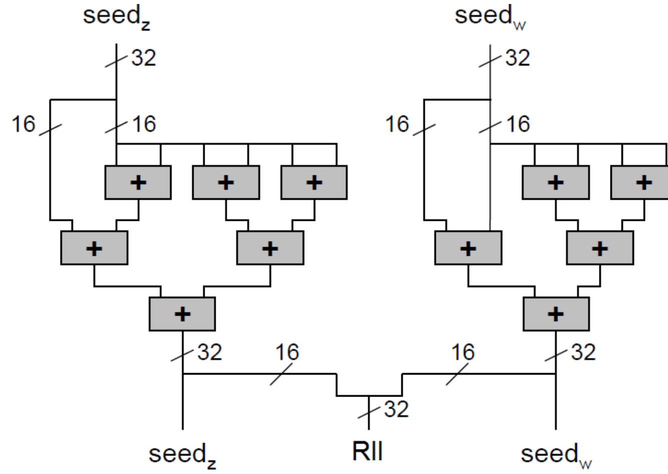
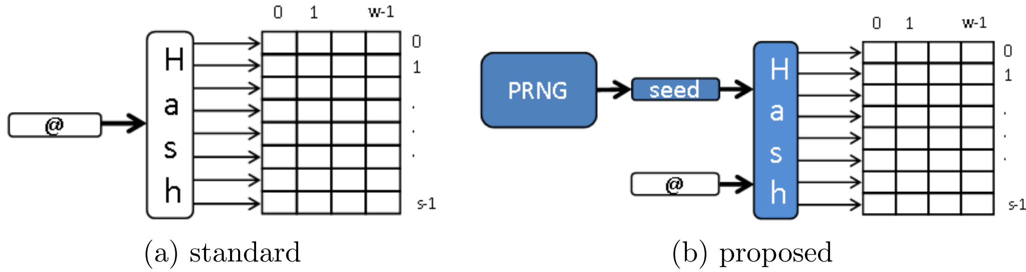


Figure 4.2: Implementation of the PTA-MWC PRNG



(a) standard (b) proposed

Figure 4.3: Block diagram of the cache design.

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 (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 4.2.

We have implemented the PTA-MWC in the CACTI tool [28], 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 [33]. 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

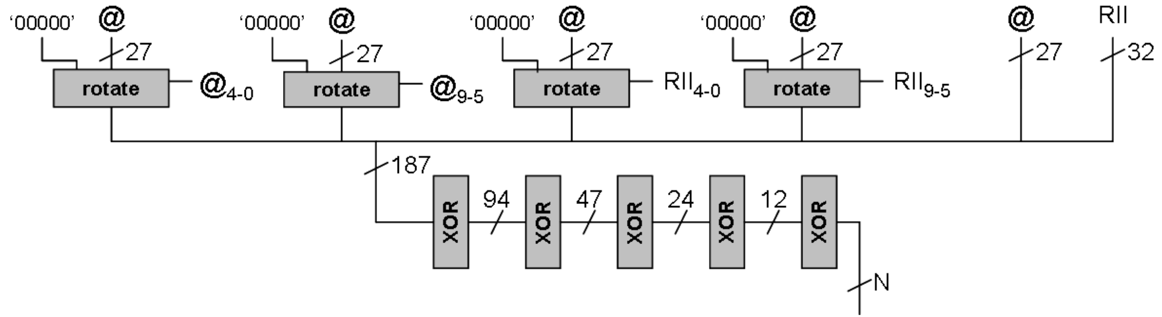


Figure 4.4: Parametric hash function proposed for the RP cache.

repeat.

4.6.2 Random Replacement Policy

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.

4.6.3 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 4.3, 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 4.4 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 4.4, 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 [20]. 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 [28]. 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.

5

Compiler and Run-Time Support for Randomisation

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 the case of software, memory objects may refer to program, functions, basic-blocks, data structures, etc. These objects can be created off-line, by the compiler and the linker, or on-line as part of the program execution by the program loader and run-time memory-related libraries.
The location at which memory objects of a program are placed into cache, i.e. the *cache layout* (see Section 4.4), is determined by the placement and replacement policies. The PROARTIS random cache design implements both random policies making cache conflict layouts, and so observed execution time, vary randomly across program invocations. This makes the PROARTIS random cache design presenting execution times fulfilling the properties required by the PTA methods. A similar behaviour to that provided by our random placement and random cache can be achieved with deterministic cache designs if an appropriate software support is provided. This section presents a compiler technique that makes programs run on top of deterministic caches to behave as if they were run on top of time randomised caches.

5.1 Random Cache Behaviour on Deterministic Caches

Given a program and a deterministic cache design, e.g. modulo placement and LRU or FIFO replacement policies, all runs of the program lead to one cache conflict, making the execution time not varying across those invocations (runs) 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 in which it is allocated and (2) its relative order with respect to the other memory objects, which determines the cache way.
In order to force deterministic cache designs to behave as time randomised, memory objects must be located into random cache positions. To do so, in theory, software must ensure that (1) memory objects are placed in random memory locations across runs and (2) the order in which memory objects are accessed is randomised.
The former makes the cache set to be randomly selected at every new memory

location, and so mimicking the behaviour of a random placement policy. The latter makes 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.
Therefore, PROARTIS has focused on software-only approaches to randomise the placement of memory objects. That is, changing the memory position in which objects are located, 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 previous sentence is weird. This makes the software approach the perfect candidate to be applied to current high performance processor designs that already implement a random replacement policy [2] [19]. The next sections describe in detail software techniques evaluated within PROARTIS to this end [1].

5.2 Software Components and Memory Objects

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

- *Program loader*. It is in charge of loading the executable program into main memory. The program loader is an operating system module that operates online. However, it can also operate off-line before the system becomes operational, being in this case not part of the operating system.

- *Run-time memory-related libraries*. They are in charge of providing services related to dynamic memory to programs.

During the second phase of the project, we have focused on functions and stack memory objects managed by the compiler. During the third phase of the project, we will focus on randomizing the location of memory objects into the heap, as well as investigating the implications of software randomization techniques with multi-core execution.

[1] The work presented in this section has been submitted as a conference paper in the Real-Time System Symposium (RTSS)

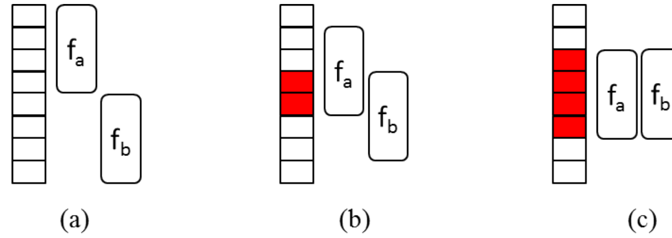


Figure 5.1: Cache location of functions f_a and f_b in a direct-mapped cache implementing modulo placement policy.

5.3 Random Location of Memory Objects

In order to elaborate our argument, we assume a direct-mapped cache implementing a modulo placement policy. Moreover, let us consider a program formed by a loop that calls two functions, f_a and f_b, each composed of sequential code. The total size of the two functions is smaller that 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 5.1 shows three different cache layouts. In Figure 5.1(a), the two functions are placed in consecutive memory positions that do not collide with each other thereby taking maximum benefit of the cache. However, if they are placed in memory positions such that the modulo function makes two pairs of addresses from the two functions collide into the same cache set, the effectiveness of the cache will be low or even null because systematic cache misses will occur. Figure 5.1(c) shows the case in which f_a and f_b are located in memory addresses that map into the same cache lines. Figure 5.1(b) shows an intermediate case in which half of each function 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 5.1(c)) occur with a given probability.

5.3.1 Memory Object Size

The size of memory objects plays an important role into the software-randomization approach. The reason is 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 with respect to the first address does not change, intra-object cache conflicts will remain the same, independently of the memory position of the object. For example, if we consider a function composed of 9 consecutive elements (each fitting within a cache line), and a direct map cache composed of 8 cache line entries, the first and the last elements of the function will always collide into the same cache line, independently of its memory location. Such a deterministic behaviour inside the object may not affect the i.i.d. properties provided by the random object location, if the size and the number of objects are selected properly. Next we discuss about it.

In order to develop our argument, let us consider a program composed of only one function, with no calls to the operating system. In this case, only one object will be

randomly allocated in memory, and so the random location will have no effect on execution time[2], not fulfilling the i.i.d. property. Instead, if we consider the same program but divided into multiple functions with a size equal to the cache line size, the random location of those functions will make the deterministic cache behaving exactly as having a hardware random placement policy, since each function, and so cache line will be located in a random cache set.
In general, the number and the size of each memory object will determine the number of cache conflict layouts generated by the random object location. Similarly to the argument provided in Section 4.4, the more objects we have, the higher are the number of cache conflict layouts and so the less probable is to end up in a bad cache conflict layout that leads to a long execution time.

5.3.2 Influence of the Deterministic Placement Policy

The PROARTIS hardware random placement policy presented in Section 4.3 has a probability of $1/S$ of selecting a cache set for a given object, where S is the total number of cache sets ($S = 8$ in the example above). Hence, in order to mimic 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. $1/S$. Unfortunately, this probability not only depends on the software methodology but also on the deterministic placement policy considered. Next we discuss this issue.
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 M/S different chunks, M being the total number of main memory entries. Within each chunk, memory addresses are mapped to the same cache set. Figure 5.2 shows a 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 locating 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 in Figure 5.2 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 placing it 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 is lower (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 $128/1020$.

[2]If the program would call operating system services, interferences with them may change at every function location

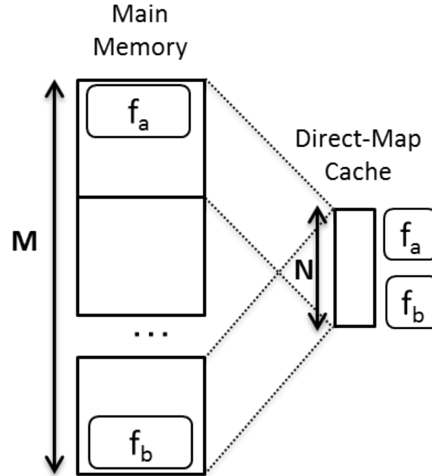


Figure 5.2: Location of functions f_a and f_b in main memory.

In conclusion, those cache sets already mapped are less probable to be selected. Such an effect can be eliminated if objects are located in a chunk basis. In other words, the remaining memory entries within a chunk are not considered to allocate new objects. By doing so, in the previous example f_b is placed into any 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 N (which is commonly the case), and so this effect can be neglected. If we consider a case in which $M = 10^6$ and $S = 32$, the biased probability differs in the fifth decimal digit (0.03125 and 0.031249) of a cache set assigned to a memory entry with respect to another that is not.

5.4 Computation of ETPs at Processor Instruction Level

PTA imposes that the timing behaviour of processor instructions must have either no dependence on the execution history or dependence that can be characterised probabilistically (see Section 2.2.2). This section shows that such an important requirement is fulfilled by the software approach as well.
The software approach allocates blocks of instructions (in case of functions) and variables (in case of stack frames) in random memory positions. This makes all elements within a block, i.e. the cache lines that form the block, to depend on the location of the first element. However, such a dependence does not break the randomisation property. That is, since the first element is randomly located, the location of the subsequent elements is random as well.
Let us consider the function f_a in Figure 5.2(a). The probability of the third element of f_a to be placed in the fourth cache set (marked in red) equals to the probability of the first element of f_a to be placed in the second cache set: $1/S$. In general, if we consider the element within the block a e_i^a, being i its relative position with respect to the first element of the block, the probability of being placed in

cache set s is: $P^a_{place}(s) = P^a_{place}(f(i,s))$, being $f(i,s)$ a function that provides the first cache set location of the beginning of the block.
Therefore, e^a_i will miss in cache if there exist an element belonging to another block b that is also located in the same cache set:

$$P_{miss}(e_i) = P_{place}b_1(s) \cdot P_{place}b_2(s) \cdots P_{place}b_z(s) \tag{5.1}$$

being z the number of blocks that have been randomly located between two calls of a. Therefore, similar to the hardware random placement policy, the random software memory placement allows to characterise probabilistically the timing behaviour of memory requests based on previous execution history, i.e. z and i.

5.5 Random Software Approach Implementation: Stabilizer

During the second phase of the project, we have focused on random memory location of function and stack frames 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 in 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 section 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 on DieHard [3]. DieHard is a memory allocator that uses heap randomisation 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.

5.5.1 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 pointed by the relocated function (see Figure 5.3). By doing this, every function call or global access in the function is indirected through the RT. Each function points 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 5.3, 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.

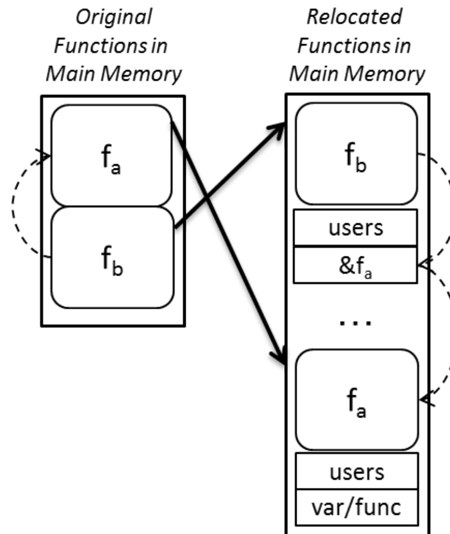


Figure 5.3: Relocation table placed immediately at the beginning of functions f_a and f_b into the main memory.

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 re-allocating functions on the fly 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. Stabi7lizer 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.

5.5.2 Stack Randomisation

Stabilizer randomises the stack by making it non-contiguous: each function call moves the stack to a random location (see Figure 5.4). 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

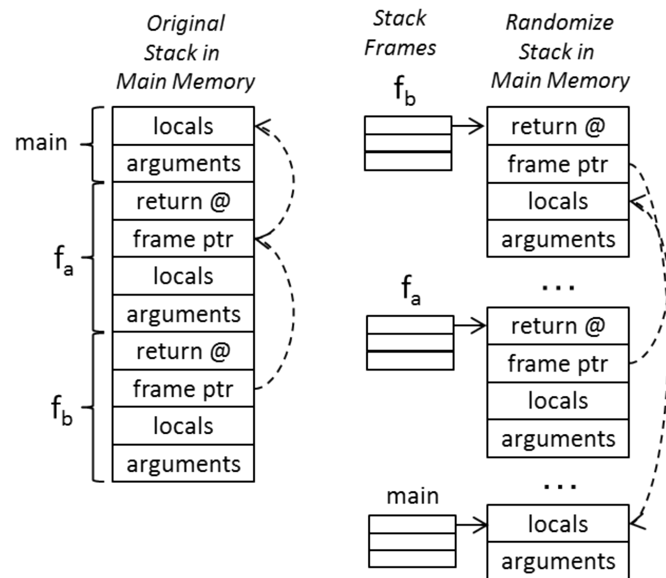


Figure 5.4: Randomisation of stack frames of functions f_a and f_b into the main memory.

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 also allows to re-allocate stack frames on the fly while the program is being executed. To do so, Stabilizer invalidates 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.

Acronyms and Abbreviations

- CDF: Cumulative distribution function.
- DM: Direct map.
- ETP: Execution time profile.
- EVT: Extreme value theory.
- FA: Full associative.
- HW: Hardware.
- ICDF: Inverse cumulative distribution function.
- MBPTA: Measurement-based probabilistic timing analysis.
- MBPTA-EVT: Measurement-based Probabilistic Timing Analysis with Extreme Value Theory.
- MWC: Multiply-with-carry.
- PDF: Probability distribution function.
- PRNG: Pseudo-random number generator.
- PTA: Probabilistic timing analysis.
- pWECT: Probabilistic worst-case execution time.
- RII: Random index identifier.
- RP: Random placement.
- RR: Random replacement.
- RT: Relocation table.
- SA: Set associative.
- SMP: Symmetric multi-processing.
- SPTA: Static probabilistic timing analysis.
- TLB: Translation look-aside buffer.
- WCET: Worst-case execution time.

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] 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.

[4] J.V. Bradley. *Distribution-Free Statistical Tests*. Prentice-Hall, 1968.

[5] A. Burns and D. Griffin. Predictability as an emergent behaviour. 2011.

[6] J.M. Cargal. *Discrete Mathematics for Neophytes: Number Theory, Probability, Algorithms, and Other Stuff*. 1988.

[7] F.J. Cazorla, E. Quiñones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. PROARTIS: Probabilistically analysable real-time systems. *ACM Transactions on Embedded Computing Systems*, to appear.

[8] R.N. Charette. This car runs on code. In *IEEE Spectrum online*, 2009.

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

[10] Andrew Coombes. How to measure and optimize reliable embedded software. In *ACM SIGAda Annual International Conference*, 2011.

[11] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzetti, E. Quinones, and F.J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *Proceedings of the 24th Euromicro Conference on Real-Time Systems*. IEEE, July 2012. To appear.

[12] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *the 22nd IEEE Real-Time Systems Symposium (RTSS01)*, pages 215–225, 2001.

[13] W. Feller. *An introduction to Probability Theory and Its Applications*. 1996.

[14] 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.

[15] Edward Frees and Emiliano Valdez. Understanding relationships using copulas. *North American Actuarial Journal*, 2:1–25, 1998.

[16] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading MA., 1988.

[17] 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.

[18] 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.

[19] http://www.arm.com. *ARM Cortex-R4 processor manual.*

[20] S. Huntzicker et al. Energy-delay tradeoffs in 32-bit static shifter designs. In *ICCD*, 2008.

[21] S. Kotz and S. Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.

[22] 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.

[23] 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.

[24] T. Lundqvist and P. Stenstrom. Timing anomalies in dynamically scheduled microprocessors. In *RTSS*, 1999.

[25] G. Marsaglia and A. Zaman. A new class of random number generators. *Annals of Applied Probability*, 1(3):462–480, 1991.

[26] E. Mezzetti and T. Vardanega. On the industrial fitness of wcet analysis. *11th International Workshop on Worst-Case Execution-Time Analysis*, 2011.

[27] Frank Mueller. Timing analysis for instruction caches. *Real-Time Systems - Special issue on worst-case execution-time analysis archive*, 2000.

[28] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. CACTI 6.0: A tool to understand large caches. *HP Tech Report HPL-2009-85*, 2009.

[29] S.M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technical University of Munich, 2002.

[30] Eduardo Quinones, Emery D. Berger, Guillem Bernat, and Francisco J. Cazorla. Using Randomized Caches in Probabilistic Real-Time Systems. In *22nd Euromicro Conference on Real-Time Systems (ECRTS)*, pages 129–138. IEEE, 2009.

[31] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, November 2007.

[32] 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.

[33] 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.

[34] 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.

[35] 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.

[36] S. Zhou. An efficient simulation algorithm for cache of random replacement policy. In *Proceedings of the 2010 IFIP international conference on Network and parallel computing*, NPC'10, 2010.