Universitat Politècnica de Catalunya

Master Thesis

Exploring average-case and probabilistic worst-case performance of time randomized caches and their associated overheads

Author:
Suzana Milutinović

Supervisors:
Dr. Jaume Abella
Dr. Francisco J. Cazorla

A thesis submitted in fulfilment of the requirements for the degree of Master of Science

Departament d'Arquitectura de Computadors
Facultat d'Informàtica de Barcelona

January 2016

Declaration of Authorship

I, Suzana Milutinović, declare that this thesis titled, 'Exploring average-case and probabilistic worst-case performance of time randomized caches and their associated overheads' and the work presented in it are my own. I confirm that:

- This work was done wholly or mainly while in candidature for a research degree at the Universitat Politècnica de Catalunya.
- Where any part of this thesis has previously been submitted for a degree or any other qualification at the Universitat Politècnica de Catalunya or any other institution, this has been clearly stated.
- Where I have consulted the published work of others, this is always clearly attributed.
- Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.
- I have acknowledged all main sources of help.
- Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: _____

Date: _____

Abstract

Probabilistic timing analysis (PTA) is a powerful approach to derive worst-case execution time (WCET) estimates, as needed in safety-critical systems, in the presence of high-performance hardware features (e.g. caches). To that end, the timing behavior of certain hardware resources, such as caches, is randomized. Time randomized (TR) caches allow deriving hit/miss probabilities for each access and probabilistic WCET estimates for the overall program.

For the static variant of PTA, called SPTA, we identify one of the main elements that jeopardizes its scalability to real-size programs in deriving WCET estimates: its high computation time cost. SPTA's high computational cost is mainly due to *convolution*, a mathematical operator used by SPTA and also deployed in many domains including signal and image processing. We show how convolution is applied in SPTA, and qualitatively and quantitatively evaluate optimizations to convolution when applied to SPTA. We show that the techniques that we have called discretization and sampling provide larger execution time reductions, at the cost of a small loss of precision.

For the measurement based variant of PTA, called MBPTA, we address the lack of efficient ways to analyze the average-case execution time(ACET) of TR caches, which is needed for low-critical high-performance tasks in mixed-criticality environments. So far, the average performance of a TR cache can only be analyzed through simulation, whose accuracy strongly depends on carrying a large number of simulations. We respond to this challenge by proposing PACO, an accurate analytical approach to estimate cache hit/miss probabilities of full applications, parts of them and individual cache accesses at low cost for a wide variety of TR cache hierarchies and setups.

Acknowledgements

I would like to express my sincere gratitude to Dr. Jaume Abella and Dr. Francisco Cazorla, my research supervisors, for their valuable guidance and engagement throughout my work on this thesis.

Futhermore I would like to thank my colleagues from the Computer Architecture/Operating System Interface research group of the Barcelona Supercomputing center, from whom I have learned a great deal over the past two years.

The research leading to these results has received funding from the European Community's FP7 under the PROXIMA Project, grant agreement no 611085. This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, the HiPEAC Network of Excellence, and COST Action IC1202: Timing Analysis On Code-Level (TACLe).

Contents

List of Figures

List of Tables

Abbreviations

ACET	**A**verage **C**ase **E**xecution **T**ime
BCET	**B**est **C**ase **E**xecution **T**ime
ETP	**E**xecution **T**ime **P**rofile
ISA	**I**nstruction **S**et **A**rchitecture
MBPTA	**M**easurement **B**ased **P**robabilistic **T**iming **A**nalysis
MBTA	**M**easurement **B**ased **T**iming **A**nalysis
PACO	**P**robabilistic **A**nalytic **C**ache m**O**delling
PTA	**P**robabilistic **T**iming **A**nalysis
SPTA	**S**tatic **P**robabilistic **T**iming **A**nalysis
TR	**T**ime **R**andomized
WCET	**W**orst **C**ase **E**xecution **T**ime

Chapter 1

Introduction

1.1 Motivation

In recent years *real-time systems* - which need to complete their work and deliver the response within a specified time constraint - have become widely spread in our daily life. Examples of applications deploying real-time technologies are automated manufacturing systems, automotive engine control systems and medical instrumentation equipment. The majority of real-time systems are *embedded*, which means that they represent a component of a larger engineering system, subject to their control.

Real-time systems commonly comprise a set of concurrent tasks, where each task issues jobs to perform computational activity conforming to a set of constraints. A common form of time constraint assigned to tasks is a *deadline*, which specifies the time before which a task must complete its execution. According to the potential consequences of missing its deadline, a task can be:

- **Hard real-time task**: all jobs of the task must complete its execution before their assigned deadline. The system is considered to have failed if any job misses the deadline.

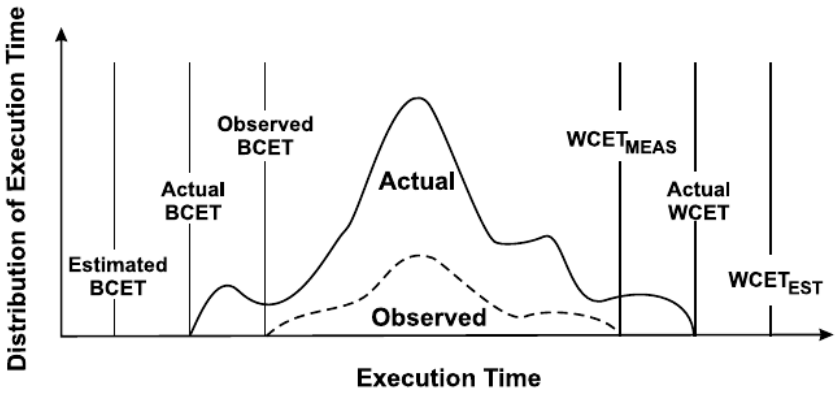


FIGURE 1.1: Execution Time distribution (Taken from [1])

- **Soft real-time task**: some of the jobs of the task may miss their deadline, which will result in a degraded quality of the outcome, but not a system failure. Typically, there is an upper bound on the number of misses that can occur during a defined interval.
- **Firm real-time task**: some of the jobs of the task may miss their deadline, the system will not fail, but it will discard late results, as they do not have any value.
- **Not real-time task**: does not have time constraints.

The same taxonomy is extended to systems, so we classify them as hard, soft and firm real-time systems, though the different tasks of the same system may belong to different categories.

In real-time systems *timing analysis* is performed to determine the execution times of tasks, which are used by a *schedulability analysis* to determine whether the system will be able to meet its timing constraints. Typically the execution time of a task varies depending on its input values, the underlying platform, the behavior of the other tasks in the system, etc., as presented in Figure 1.1 by Actual distribution. The shortest execution time of a task is called the *best-case execution time (BCET)*, while the longest one is the *worst-case execution time (WCET)*. *The average-case execution time (ACET)* lies somewhere in-between the BCET and the WCET.

Timing analysis in real-time systems traditionally has been concerned with computing the WCET of a task, to ensure that it will complete execution before its deadline. With a limited number of measurements during the test phase, we typically observe only a limited part of the Execution Time distribution, shown in Figure 1.1 as Observed distribution. Capturing the whole (actual) distribution - and observing the actual WCET - would require repeated measurements for all possible execution paths and hardware states. Therefore, timing analysis provides techniques to estimate or bound the WCET.

The WCET estimate/bound provided by the timing analysis has two requirements: 1) *reliability* - it actually upper bounds all possible execution times; 2) *tightness* - it should be close to the actual WCET to avoid costly over-dimensioning. As hardware platforms employed in real-time systems increase in complexity, satisfying both conditions becomes increasingly challenging [2].

Timing analysis techniques are generally classified as static and measurement-based [2]. Static timing analysis estimates WCET from an application code and an abstract model of a hardware, without executing a program. Measurement-based timing analysis estimates WCET by measuring the execution time of the whole program or program fragments (the latter also analyzes the structure of a program in which case it is referred to as hybrid analysis). The longest observed execution time is taken and scaled by an ad hoc *engineering margin* (e.g. 20%), which is not scientifically backed, but based on user experience and knowledge of the hardware and software.

Probabilistic Timing Analysis (PTA) [3–9] has emerged recently as a way to achieve predictability through probabilistic means instead of through deterministic means, as most timing analyses do [10]. Instead of a single WCET estimate, PTA techniques provide a distribution of WCET estimates that can be exceeded with a given – arbitrarily low – *probability*, which are typically referred to as probabilistic WCET (pWCET) estimates, see Figure 1.2. Similarly to traditional (deterministic) timing analysis, PTA techniques are classified as *static (SPTA)* or *measurement-based (MBPTA)*.

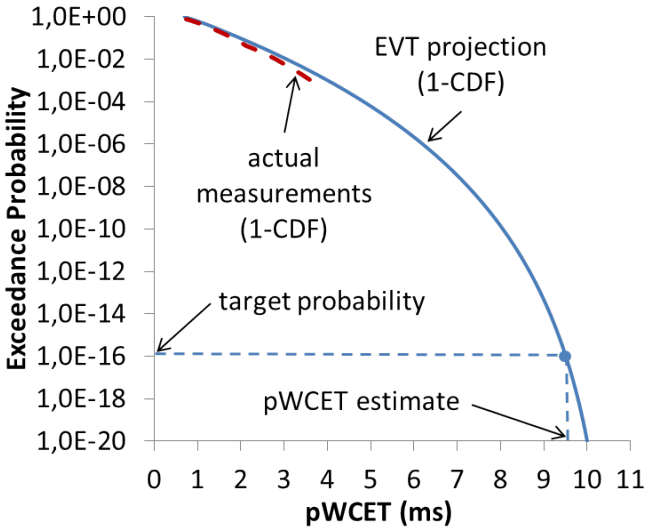


FIGURE 1.2: Example of $CCDF$ and pWCET

Under the static variant of PTA, each instruction has a probabilistic timing behavior represented with an *Execution Time Profile (ETP)*. An ETP is expressed by a timing vector that enumerates all the possible latencies that the instruction may incur, and a probability vector, which for each latency in the timing vector, lists the associated probability of occurrence. Hence, for an instruction $\mathcal{I}_i$ we have $ETP(\mathcal{I}_i) = <\vec{t_i}, \vec{p_i}>$ where $\vec{t_i} = (t_i^1, t_i^2, ..., t_i^{N_i})$ and $\vec{p_i} = (p_i^1, p_i^2, ..., p_i^{N_i})$, with $\sum_{j=1}^{N_i} p_i^j = 1$. The *convolution* function, $\otimes$, is used to combine ETPs, such that a new ETP is obtained which represents the execution time distribution of the execution of all the instructions convolved (i.e. all the instructions of the program under analysis).

The measurement-based variant of PTA carries out end-to-end runs of the program on the target hardware. MBPTA applies Extreme Value Theory (EVT) [11], a well-known statistical method, that builds upon the complementary cumulative distribution function (CCDF or 1-CDF) of the observed execution times to project the probability distribution of the execution time of a given run of a program to exceed a threshold pWCET, see Figure 1.2. MBPTA is close to industrial practice to compute WCET estimates and has been evaluated for avionics [7] and automotive [12] setups.

PTA techniques pose some requirements on hardware and software designs [13],

that must either have: 1) fixed latency with no jitter (or upper bounded jitter at analysis); or 2) randomized timing behavior. One of the hardware resources whose timing behavior is randomized to enable PTA are cache memories. Caches have been an object of intense study in the real-time domain [14–20], due to the complexity they introduce to WCET analysis. In this line, a study conducted for the European Space Agency [20] shows the difficulties of using caches since small program changes that lead to different memory layouts can trigger pathological cache behavior which were called cache risk patterns. The complexity in the analysis of the cache lies on the fact that caches are stateful resources so whether an access hits or misses depends on the location of memory objects (which easily changes across different runs), which determines their cache set placement, and on the sequence of cache accesses.

Time randomized (TR) caches [21, 22] employ random replacement and random placement policies. They break the dependence between memory location of the objects accessed and cache placement such that the hit/miss probability of an access is not affected by the particular memory address accessed, as needed by PTA [21]. TR caches have been shown to be competitive in terms of worst-case performance with respect to standard time-deterministic caches deploying modulo placement and least-recently-used replacement [10]. A wide variety of TR cache configurations and hierarchies have been proven analyzable in the context of measurement-based PTA (MBPTA) [6, 7]. Those configurations include multi-level caches; direct-mapped, fully-associative and set-associative caches; shared caches for instructions and data; write-through and copy-back write policies; etc.

1.2 Objectives

The static variant of PTA has recently received significant attention [5, 8, 9, 23]. This thesis contributes to SPTA development by identifying and mitigating one of the major bottlenecks for SPTA to scale to industrial-size programs: its execution time requirements. With real-time programs growing in size, the need to carry out

a convolution operation for every instruction in the object code may incur high computation time requirements. Hence, efficient ways to perform convolutions in the particular context of SPTA need to be found.

In the context of the measurement-based variant of PTA previous work focused on exploring worst-case performance. While not used in schedulability analysis, the average performance is critical to improve other non-functional metrics (e.g. power consumption) in real-time systems. Also, in recent years, there is an increasing trend of integrating multiple functionalities upon a single platform. In the general case, these functionalities may have different levels of importance or *criticalities* - the systems with this property are called *mixed-criticality* systems. Some current real-time mixed-criticality systems comprise both real-time and high-performance applications, which makes average performance (and computing ACET) an important factor in the real-time domain. For instance, in the space domain it is well accepted that systems will be dual-criticality [24] with control applications requiring real-time guarantees, hence designed to meet requirements in the worst case; and high-performance payload applications requiring high average performance.

Since caches have very high impact on average performance, a fast evaluation of different cache setups becomes critical in the design of a system. However, exploring the cache design space in a tractable manner with an increasing number of interacting parameters remains an open problem due to the high number of detailed simulations required. The problem exacerbates in the context of TR caches. This occurs because, while for non-randomized caches one run may suffice to determine their average performance under a given design, for TR caches several runs are required to obtain a representative execution time distribution for each cache design point. This is particularly critical in early processor design stages to (1) design the cache architecture of a given processor for real-time systems; (2) evaluate how reference applications behave on that cache setup; and (3) tune applications to be run on that architecture. This thesis contributes to MBPTA development by proposing an approach for fast evaluation of TR caches.

1.3 Contributions

Contribution 1. We analyze a set of optimizations of the convolution operation, used in the context of SPTA. Some optimizations keep precision, whereas some others sacrifice some precision to reduce computational cost, while preserving the fact that the result is still a trustworthy WCET estimate for the program under analysis.

- Among precision-preserving optimizations we consider convolution parallelization, which has been largely studied previously in the literature [25, 26], in two forms: (1) *inter-convolution parallelization*, where ETPs to be convolved are split into several groups that are convolved in parallel and (2) *intra-convolution parallelization* where one (or both) of the ETPs to be convolved is split into sub-ETPs so that each sub-ETP is convolved with the other ETP in parallel.

- Among optimizations that sacrifice some precision to reduce convolution cost, we propose (3) *discretization*, such that few different forms of ETPs exist and convolutions across identical ETPs don't need to be carried out too often. We also propose (4) *sampling* where several elements in the ETP are collapsed into one [27], thus reducing the length of the ETPs to be convolved and so the number of operations.

Our results show that discretization and sampling lead to the highest reductions in execution time, whereas the combination of intra-convolution and inter-convolution parallelization provides second order reductions in execution time. In particular, discretization and sampling reduce execution time by a factor of 10 whereas precision-preserving optimizations reduce it by a factor of 2, thus leading to a total execution time reduction factor above 20 so that execution time is below 5% than with the original convolutions. This execution time reduction comes at the expense of a pWCET increase around 3%.

As a result of this work we have published the following paper: *Suzana Milutinovic, Jaume Abella, Damien Hardy, Eduardo Quiñones, Isabelle Puaut, Francisco J. Cazorla, "Speeding up Static Probabilistic Timing Analysis", in proceedings of the 28th GI/ITG International Conference on Architecture of Computing Systems (ARCS), Porto (Portugal), March 24-27 2015.*

Contribution 2. We propose a new method to evaluate the ACET of TR caches in a quick manner, called ***P**robabilistic **A**nalytic **C**ache m**O**delling (PACO)*. Given a set of reference programs from which we extract a trace of instruction and data memory accesses, PACO derives tight estimates of cache miss probabilities, P_{miss}[1]. We propose tight approximation formulas, which we implement in PACO, to estimate P_{miss} for every access in the program as a way to understand the performance of programs running on top of TR caches, thus removing the need for carrying out a large number of time-consuming cache simulations. PACO builds upon the formulas used in the context of MBPTA [21, 22], whose purpose was simply illustrating MBPTA compliance of those cache designs. PACO extends formulas to a wide variety of cache setups and improves their accuracy.

- We assess the accuracy of existing formulas [21, 22] approximating P_{miss} for several cache hierarchies and configurations. We do so by comparing the approximated probabilities obtained analytically and the more accurate probabilities obtained with a very large number of simulations.
- We identify some sources of inaccuracy for a number of formulas due to their inability to capture dependencies across random events in the context of TR caches.
- We deliver some new approximations for some of those formulas proving that higher accuracy can be achieved.

PACO has an overall inaccuracy below 2.6% across all cache configurations. PACO's execution time is comparable to that of running only 10 simulations per cache

[1]The probability of hit (P_{hit}) is given by $P_{hit} = 1 - P_{miss}$.

design-point, much less than needed to obtain stable (representative) average performance estimates for TR caches.

As a result of this work we have published the following paper: *Suzana Milutinovic, Eduardo Quiñones, Jaume Abella, Francisco J. Cazorla, "PACO: Fast Average-Performance Estimation for Time-Randomized Caches", in proceedings of the 52nd ACM/IEEE Design Automation Conference (DAC), San Francisco (California), June 7-11 2015.*

1.4 Organization

The rest of this thesis is organized as follows:

- Chapter 2 provides some background on cache designs, on timing analysis of real-time systems and on the main characteristics of SPTA and MBPTA relevant for this thesis.
- Chapter 3 presents our contributions to reduce the computational cost of SPTA together with a set of experimental results.
- Chapter 4 introduces our model towards estimating analytically the average performance on top of TR caches and evaluates empirically its accuracy.
- Finally, Chapter 5 summarizes the main conclusions of this thesis and points towards interesting future works.

Chapter 2

Background and related work

2.1 Cache memories

In recent years processor and main memory speed have been increasing at different rates, making the accesses to the main memory a major performance bottleneck in modern systems. The problem exacerbates with the shifting of industry toward multicores, as memory needs higher bandwidth capacity to respond to the requests from multiple cores. One of the mechanisms introduced to mitigate these bottlenecks are cache memories.

Cache memories are small and very fast, typically SRAM-based memories that temporarily store copies of data from main memory likely to be used soon by the processor. The motivation for caches comes from two properties of program code and data:

- *Temporal locality*: memory words accessed in the past are likely to be accessed again (e.g. accesses in loops).
- *Spatial locality*: a future access is likely to be in the nearby location to the past one (e.g. iterations through vectors)

On an access to a memory address, first the caches are searched. If the datum at the requested memory location is found in cache, it is returned with low latency (the event is referred to as a *cache hit*). Otherwise, the datum is brought from the main memory (or the next cache in a hierarchy, as explained in Section 2.1.3) and returned with higher latency (the event is called *cache miss*). Typically, miss latencies are one or more orders of magnitude higher than hit latencies.

To exploit spatial locality, a portion of data is transferred from memory to cache (and stored in cache entries) on a cache miss rather than the single datum requested. This group of several words on consecutive memory addresses is called *cache block* or *cache line*. The common size of cache lines nowadays may range between 16 and 128 bytes.

Along with the data block, each cache entry keeps its unique identifier called *tag* and some state bits. As cache size is much smaller than main memory size, several data blocks are mapped to the same cache entry - the tag is used to differentiate them. Depending on the cache organization, only one or few of those data blocks mapped to the same cache entry (or set of cache entries) can be stored simultaneously in cache. The state bits are used for cache management. Two most common ones are the *valid bit*, which indicates if the cache entry contains valid data, and the *dirty bit*, which indicates if the data block was modified since it was loaded to the cache entry and such modification was not propagated to the following caches in the cache hierarchy (or main memory).

2.1.1 Cache organization

Depending on how data blocks are mapped to the cache entries, we differentiate three common cache organizations: *direct mapped*, *fully associative* and *set associative*. In a sense, the cache can be seen as a bidimensional array with a number of rows (sets, S) and a number of columns (ways, W). Each entry in the table is a cache line (cache entry) consisting of a fixed number of bytes (B). Typically, all parameters (sets, ways and bytes per cache line) are a power-of-two.

In direct mapped caches a data block can only be placed in a single entry in cache, so they consist of a single column (1 way). They provide fast access and simple implementation, since on a cache access a single entry needs to be checked to know whether a cache hit or miss occurred. On the negative side they suffer from a lot of conflicts since two cache blocks mapped to the same entry cannot coexist in cache.

In fully associative caches a data block can be placed in any entry in cache, so they consist of a single row (1 set). Due to their flexibility they usually provide the highest hit rate, but they are slow and complex in design since, on a cache access, all cache entries need to be looked up.

Set-associative caches emerge as a combination of both direct mapped and fully associative caches in an intent for combining the advantages of both: low access latency and high hit rates. Set-associative caches are organized as a group of direct mapped caches with identical size, so they consist of a number of sets and ways. A given data block can only be placed exactly in one cache entry in each way. On a cache access, those cache entries (as many as cache ways) need to be looked up. There is a clear tradeoff between the access latency (lower for a lower number of cache ways) and the hit rate (typically higher for a higher number of cache ways).

In the case of direct mapped and set associative caches, the mapping of a data block to a corresponding cache entry/set is defined by a placement policy (more details are provided in Section 2.1.2).

Typically caches are direct-mapped or n-way set associative, where n is a small number (≤ 8). Fully associative caches are used only for some specialized memories, like translation look-aside buffer whose size is small enough to keep access latency low.

2.1.2 Cache management

Placement policy. The placement function determines the cache set in which a data block is stored. The most common policy is *modulo function* which, assuming

that the number of sets in cache is a power-of-two ($S = 2^s$), extracts s bits from the memory address being accessed and uses them as an index. Though easy to implement, the main limitations of this scheme is that addresses are systematically placed across cache sets. As a consequence certain memory access patterns may lead to repeated misses in the cache. For instance, addresses separated by $k \cdot S \cdot B$ bytes in memory where $k \in \mathbb{N}$ and $S \cdot B$ is the size of one cache way, are mapped to the same cache set systematically.

Replacement policy. To be able to load a new data block to cache, it is often required to evict another one that was fetched in the past due to limited size of caches. The replacement policy determines how to choose a line for eviction, called *victim*, in the set where the data block being fetched is mapped. In the case of direct mapped caches, there is only one choice - the only line in the cache set where the block is mapped. For fully associative and set associative caches several policies are proposed, among which the most popular is LRU (least recently used).

Write policy. When a data block is loaded in cache, there are at least two existing copies of it: one in cache, another in memory and, potentially, others in other cache memories in the hierarchy. The write policy determines whether on a write operation only the copy in this cache is updated or it is also updated the copy in the next level in the memory hierarchy (either another cache or main memory). If the next level is systematically updated, then we talk about *write-through* policy. If only the copy in cache is updated, then we talk about *copy-back* policy.

When copy-back policy is used, once a cached copy is chosen for replacement, the copy in the following level in the memory hierarchy needs to be updated. A special dirty bit is used to track whether the cache copy has been modified since it has been loaded in cache. Write operations in the next memory level will occur only if the data block is marked as dirty.

Write allocation. When a data block is not loaded in a cache, one may choose between two actions on a write operation: to load the block in the cache and modify it - *write-allocate* policy - or to modify block directly in memory without fetching it to the cache - *no-write allocate* policy. Write-allocate is commonly used with copy-back designs, whereas no-write allocate is used with write-through designs.

2.1.3 Multi-level cache hierarchy

Choosing an appropriate cache size is an important design issue. While increasing the size of the cache improves the hit rate, it results in longer hit latency and higher power consumption. This trade-off is addressed by building a cache hierarchy with multiple levels, where small fast caches are backed up by larger, slower caches. First data is looked up in the first level cache, if not found next level is checked, until data is found or eventually loaded from main memory.

Another design choice is whether to use the same cache for data and instructions. As memory regions that store code and data are commonly independent of each other, using separate - *split* - first level caches allows segregating instruction and data accesses (thus decreasing the number of cache ports) without jeopardizing cache consistency. The other choice is *unified* cache, which is typically the choice for second level caches and beyond due to the more effective use of the cache space and the fact that data and instruction accesses occur seldom so a single port can satisfy both. Using multi-level and split caches gives flexibility at choosing different cache policies.

The typical cache hierarchy nowadays employs: two small first level caches, one for data (DL1) and one for instructions (IL1), optimized for speed; a larger second level cache (L2) optimized for hit rate, usually unified; and sometimes an even larger third level cache (L3).

Inclusion property. When a cache block is loaded to a cache level, it might impact how other levels of cache are managed, as determined by the inclusion property. We differentiate inclusive, exclusive and non-inclusive cache hierarchies.

In an inclusive cache hierarchy, all lines from a certain cache are duplicated in the next level in the hierarchy. This simplifies the management when other cores in a multicore system want to remove a cache line - only last level cache is checked to determine whether a line is present. But it limits the overall capacity of a hierarchy to the size of the last level cache since cache contents in the lower (smaller) levels also exist in the higher (larger) levels.

In an exclusive cache hierarchy (very uncommon) only one copy of a line exists in the whole hierarchy. While it maximizes overall cache capacity, searching for a cache line might lead to checks on all levels, and fetching a data block into a cache level may create a number of cascade effects to ensure that no cache line is replicated.

In a non-inclusive cache hierarchy there are no restrictions nor guarantees on data duplication.

2.1.4 Time randomized caches

A time randomized cache, shown in Figure 2.1, employs random replacement and random placement policies.

We use Evict-on-Miss (EoM) as Random replacement (RR) policy, under which, on the event of a miss in a given set, a victim line in that set is randomly selected to be evicted.

Random placement (RP) [21] uses a random number, called *random index identifier (RII)*, generated either by hardware or software, and the address being accessed as its inputs. A hash function combines both and provides a unique and constant cache set (mapping) for the address along the execution. It is noted that if the RII changes, the cache set in which the address is mapped changes as well,

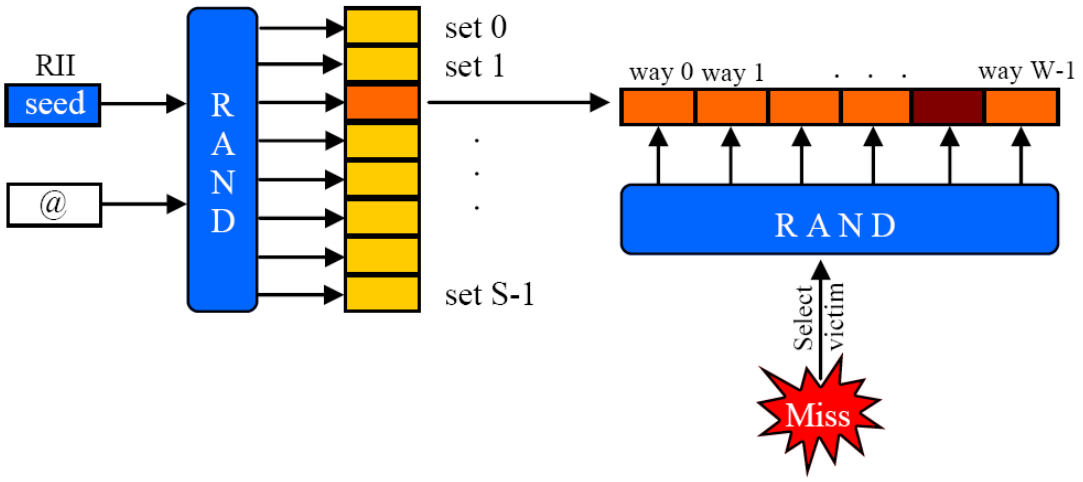


FIGURE 2.1: Time randomized cache

so cache contents must be flushed for consistency purposes. Changing the RII at program execution boundaries reduce flushing overheads. The RP policy proposed in [21] ensures that, given a memory address and a set of RIIs, the probability of mapping such address to any given cache set is the same and independent from other addresses. Therefore, the particular memory location of a given data block in memory is irrelevant for each hit/miss probabilities, thus removing systematic pathological cache access patterns.

2.2 Timing analysis and its challenges

WCET computation is an important research topic in the real-time community. A number of methods have been proposed to respond to the challenge of deriving both reliable and tight WCET estimates, with limited cost and effort. However, each of these methods relies on certain assumptions on the timing behavior of the system, while in reality, ascertaining these assumptions may be overly difficult or impossible [28].

Static timing analysis derives WCET estimates without running a program - it analyzes the program structure to determine the possible program flows (*high-level analysis*) and construct a model of a hardware to determine the execution time

of instructions (*low-level analysis*). To be able to cover all possible input values and states, static methods rely on abstractions for both hardware (e.g. abstract hardware states) and software (e.g. contexts of execution).

The main limitation of the static approach is that the correctness of the provided estimates depends on the availability of information needed to build an accurate timing model and information on the program flow - *flow facts*. The timing model may be inaccurate due to the inaccuracies introduced along the typical hardware development process or intentionally, due to confidentiality of information it may reveal. The flow facts may be inaccurate due to the change of the control flow during compilation, or inaccuracy in user annotations which are commonly required.

Measurement-based timing analysis derives WCET estimates based on measurements on top of the real hardware platform. The user needs to provide the stressful, high-coverage input data. The longest execution time is recorded and the WCET estimate is computed by adding an engineering margin to make safety allowances for the unknown.

The correctness of WCET estimates derived with the measurement-based approach is challenged by multiple factors: the platform used at analysis time has to be identical to the one used at deployment, producing the worst-case input is overly difficult as the input space of a program is enormous, typically there are no guarantees on the validity of the engineering margin, etc.

2.3 Probabilistic timing analysis

The main weakness of traditional (also known as *deterministic*) timing analysis techniques is that they require detail knowledge on hardware and software to compute WCET estimates. If some information about the system behavior is missing, the analysis needs to assume the worst-case, leading to potentially pessimistic (overestimated) WCET estimates, that further results in a waste of computing

resources. Acquiring a detail knowledge on hardware/software behavior in modern systems (with multi-level cache hierarchy, pipelines, branch prediction, etc.) is very difficult and costly, if at all possible. The problem exacerbates in the presence of *timing anomalies* in the system, that occur when a local non-worst-case event (e.g. cache hit) may lead to the WCET globally.

Probabilistic timing analysis technology has evolved during the last decade [3–9] and has been proven a strong competitor for traditional timing analysis techniques [2]. PTA aims to overcome the limitations of deterministic timing analysis, by reducing the amount of information needed to produce reliable and tight WCET estimates. It employs certain probabilistic analysis techniques, such as extreme value theory, to provide WCET estimates that are guaranteed to be exceeded with a specified, arbitrarily low, probability. Applying probabilistic analysis techniques on real-time systems is possible if the timing behavior of hardware/software components is constant (or upper-bounded to the highest latency) or is probabilistic having the same probability distribution at analysis and during operation, which is not a case with conventional, deterministic architectures. Thus, PTA advocates for novel, PTA-friendly architectures, in which resources are either time randomized or forced to take their worst-case latency.

Some previous work related to PTA has used different, inconsistent, terminology: *stochastic analysis* [29], *statistical analysis* [30], *probabilistic analysis* [31, 32] and *real-time queuing theory* [33]. The first paper that proposed a method based on extreme value statistics to model the worst-case execution time is [34], which was later improved in [4]. The authors in [35] point out the limitations of previous solutions - they apply extreme value theory in the general case. To be able to apply EVT correctly, platform and data need to meet certain statistical properties, which is addressed in [36]. Some other previous work has used extreme value theory in computing systems in a different context, such as task scheduling [37].

2.3.1 SPTA

In static probabilistic timing analysis, execution time probability distribution for each instruction (ETP) is determined statically from a model of the system. Along a given execution path, assuming that the probabilities for the execution times of each instruction are independent, SPTA is performed by deploying the discrete convolution ($\otimes$) of ETPs that describe the execution time for each instruction along that path. The final outcome is a probability distribution representing the timing behavior of the entire execution path. For the sake of clarity we keep the discussion at the level of a single execution path.

More formally, if $\mathcal{X}$ and $\mathcal{Y}$ denote the random variables that describe the execution time of two instructions x and y, the convolution $\mathcal{Z} = \mathcal{X} \otimes \mathcal{Y}$ is defined as follows: $P\{\mathcal{Z} = z\} = \sum_{k=0}^{k=+\infty} P\{\mathcal{X} = k\}P\{\mathcal{Y} = z - k\}$. For instance if an instruction x is known to execute in 1 cycle with a probability of 0.9 and to execute in 10 cycles with a probability of 0.1 and an instruction y has an equal probability of 0.5 to execute in 2 or 10 cycles, we have:

$$\begin{aligned} \mathcal{Z} = \mathcal{X} \otimes \mathcal{Y} &= (\{1, 10\}, \{0.9, 0.1\}) \otimes (\{2, 10\}, \{0.5, 0.5\}) \\ &= (\{3, 11, 12, 20\}, \{0.45, 0.45, 0.05, 0.05\}) \end{aligned}$$

ETPs can be defined for each static or dynamic instruction in the program. Static instructions are those present in the binary. If ETPs are defined for static instructions, then they must safely upper-bound all dynamic instances of the instruction. For instance, the ETP of an instruction in a loop must upper-bound its timing behavior in all iterations. Alternatively, ETPs can be defined for each dynamic instruction. For instance, we can define an ETP for each execution (dynamic) of an instruction in a loop. SPTA requires that the ETP of each instruction is not affected by the execution of previous instructions. If it is not possible to achieve independence across ETPs at the instruction level, another possible solution is to

compute ETPs for groups of instructions so that we obtain the ETP of their combined execution. If the ETP obtained for the group of instructions is independent, it can be convolved with the other ETPs normally.

2.3.2 MBPTA

In measurement-based probabilistic timing analysis, execution times are collected by means of measurements and extreme value theory is applied to predict the high execution times (belonging to the tail of the distribution). Based on the provided measurements, EVT estimates the parameters of the distribution that best fit the tail of interest. The WCET estimate is the value from the distribution corresponding to the chosen exceedance probability.

The critical decision for MBPTA is how to select the sample of execution times provided to the extreme value theory method. EVT requires that execution times are described with a truly random variable, and so that the execution times sample used for EVT passes some statistical tests proving its independence and identical distribution. Secondly, the sample needs to be *representative* of the target population, to compute reliable WCET estimates.

In order to produce a representative sample, it is important to identify all *sources of execution time variability (SETV)*, such as memory placement or input data, and control their influence during the collection of the sample used by EVT. This is achieved by enforcing the hardware to have the required properties and by collecting measurements conveniently. For instance, the impact of some SETV is upper-bounded by enforcing a proper initial state when collecting measurements: e.g., one may remove the impact of initial cache state on execution time by enforcing an empty initial cache (in systems without timing anomalies). Other SETV are controlled by using proper MBPTA-compliant hardware. This can be done by enforcing worst case timing behavior or time randomization of the hardware/software component. For instance, variable latency functional units (e.g., a divider) are enforced to operate at their worst latency regardless of the values operated.

Conversely, cache memories use random placement and replacement policies so that cache hit/misses occur with a probability independent of the memory location of the different program objects (code, data, stack, etc.).

To get valid WCET estimates with MBPTA it is enough to capture the effect of each SETV across different runs, while traditional MBTA methods require the tests to trigger the combination of all worst SETV during a single execution. Time randomized platforms improve the effectiveness of MBPTA, as they reduce the number of SETV that has to be controlled by the user during the collection of measurements to produce reliable WCET estimates. For further details on the application of MBPTA we refer the interested reader to the work in [36] and [38].

2.3.3 Timing analysis of cache memories

Each access to a cache memory may result in two possible outcomes: cache hit and cache miss, where these events have latencies that may differ in multiple orders of magnitude. This introduces a high variability in the execution time of cache access, and bounding it by assuming always a cache miss is overly pessimistic. Therefore, timing analysis needs to determine the outcome of each cache access, which depends on multiple factors: initial cache state, previous accesses to the cache - their order and memory addresses being accessed - and cache organization and management policies.

In the case of PTA, one needs to determine the probabilities of hit/miss in a cache. The ETP of a memory operation in a simple single-level cache hierarchy is as follows: $ETPmemop =< (l_{hit}, l_{miss}), (p_{hit}, p_{miss}) >$, where l_{hit} and l_{miss} are the hit and miss latencies respectively and p_{hit} and p_{miss} their corresponding probabilities.

P_{miss} for TR caches has been studied from different angles for both SPTA and MBPTA. It is worth noting that MBPTA only needs probabilities to exist, i.e. cache accesses need to have a probabilistic nature, but does not need to determine the actual probabilities. To that end, in the context of MBPTA only *approximation*

formulas to P_{miss} have been given. Instead SPTA [5, 8, 9] needs those probabilities to apply *convolution* across ETPs of instructions. In general, cache accesses are not independent and whether an access hits or misses impacts the probability of hit/miss of the following accesses. However, SPTA requires independence across ETPs to apply convolution. Thus, for SPTA *upper-bound formulas* to P_{miss} have been derived since they are needed to have independence for WCET estimation purposes given that any dependence cannot lead to higher P_{miss} values than those already had in the ETPs.

2.3.3.1 Miss probability for TR caches under SPTA

SPTA is intended to provide a WCET distribution upper-bounding the actual execution time distribution of the program, thus it needs P_{miss} used during timing analysis to match or upper-bound the real probability of miss once the system is deployed. Moreover, convolution operator used in SPTA requires independence across ETPs to be applied.

When time randomized caches are used, there is an intrinsic dependence among the hit probability of an access (P_{hit}) and the outcome of previously executed cache accesses [5, 8]. Existing techniques to break this dependence create a lower bound function to P_{hit} (so an upper bound to P_{miss}) of every instruction to make it independent – for WCET estimation purposes – from previous accesses [5, 8, 9].

Without loss of generality and for the sake of this explanation, we assume that each address corresponds to a different cache line. We use capital letters, e.g. A, to refer to (cache line) addresses. Whenever a subindex is added, e.g A_i, it refers to the i-th access to address A. The superindex is the absolute access count number in the considered sequence. For instance in our *reference sequence*: $(A_{j-1}, B_1^1, B_2^2, C_1^3 ..., F_1^k, A_j)$ we focus on deriving P_{miss} for a given access A_j based on the accesses carried out since the last access to A, A_{j-1}. We generically refer to the i-th access between A_{j-1} and A_j as X^i. For instance, X^3 corresponds to C_1^3, i.e. the first access to address C.

For a fully-associative cache with W ways, for the reference address sequence in which no access X^l, with $1 \leq l \leq k$, accesses cache line A, the following upper-bound can be used [9]:

$$P_{miss_{A_j}}(W) = \begin{cases} 1 - \left(\frac{W-1}{W}\right)^k & \text{if } k < W \\ 1 & \text{otherwise} \end{cases} \tag{2.1}$$

It has been shown that P_{miss} approximation formulas for MBPTA [21, 22, 39], despite being exact for some specific access sequences and upper-bounds for some others, do not provide the independence across ETPs needed by SPTA. Thus, they may lead to optimistic – so non-trustworthy – WCET estimates in the context of SPTA. As a result, SPTA relies on its own set of upper-bound probabilities that provide independence across ETPs (instructions). For instance, SPTA requires for each cache access an estimate that upper-bounds its miss probability regardless of whether previous accesses hit/miss in cache, as it is the case of Equation 2.1.

While upper-bound formulas to P_{miss} are interesting from a WCET perspective, they are of little interest from an average case perspective as they can be inordinately pessimistic with respect to the average case.

2.3.3.2 Miss probability for TR caches under MBPTA

For MBPTA *approximation formulas* to P_{miss} are used as a way to illustrate the probabilistic nature of the events occurring in TR cache organizations. For instance, for our reference sequence the miss probability on a fully-associative cache where no constraint is placed on the miss probability of X^l (where $1 \leq l \leq k$), is approximated as follows:

$$P_{miss_{A_j}}(W) = 1 - \left(\frac{W-1}{W}\right)^{\sum_{l=1}^{l=k} P_{miss_{X^l}}} \tag{2.2}$$

However, other applications of P_{miss} approximation formulas to measure average performance and, more importantly, their accuracy have not been studied yet.

We cover this gap by proposing PACO, which relies on those approximations and improves them to compute P_{miss} for instructions, data sequences and full programs in Chapter 4.

Chapter 3

WCET in Static Probabilistic Timing Analysis

3.1 SPTA performance issues

We have identified convolution of Execution time profiles (ETPs) as the most costly operation in Static probabilistic timing analysis.

Convolution complexity. The canonical form of convolution includes three steps:

- *Convolution step*: Adding latencies and multiplying probabilities for each pair of elements ⟨latency, probability⟩ from both ETPs, as illustrated in Algorithm 1 (*etpr* is a result vector after convolution of vectors *etp1* and *etp2*).
- *Sorting step*: Sorting elements in the result vector with respect to their latencies from lowest to highest.
- *Normalization step*: Combining elements with identical latencies in the result vector by adding their corresponding probabilities, as shown in Algorithm 2 (etp_{in} is the outcome of the two ETPs convolved with sorted elements and

etp_{out} is its normalized form). Since elements are previously sorted, repeated latencies may occur among consecutive elements.

Algorithm 1 Convolution step

```
1: c ← 1
2: for i = 1 to N do
3:    for j = 1 to N do
4:       etpr.lat[c] ← etp1.lat[i] + etp2.lat[j]
5:       etpr.prob[c] ← etp1.prob[i] ∗ etp2.prob[j]
6:       c ← c + 1
7:    end for
8: end for
```

Algorithm 2 Normalization step

```
1: c ← 0
2: etp_out.lat[0] ← etp_in.lat[0]
3: for i = 1 to N do
4:    if etp_in.lat[i] = etp_in.lat[i − 1] then
5:       etp_out.prob[c] ← etp_out.prob[c] + etp_in.prob[i]
6:    else
7:       c ← c + 1
8:       etp_out.lat[c] ← etp_in.lat[i]
9:       etp_out.prob[c] ← etp_in.prob[i]
10:   end if
11: end for
```

To compute timing complexity of canonical convolution, let us assume that both ETPs being convolved have size N. The convolution step requires N^2 operations to derive the probability and timing vectors, therefore it has a quadratic complexity ($\mathcal{O}(N^2)$). In the next step the resulting vector of N^2 elements needs to be sorted. While in general sorting has a cost in the order of $\mathcal{O}(M \log M)$ for a vector with M elements, the resulting ETP vector can be divided into N parts of N elements, each of them internally sorted. Thus, in practice the cost of sorting can be reduced down to linear complexity ($(\mathcal{O}(M))$). In our case, $M = N^2$, the size of the resulting vector after the convolution step. Finally, normalization step has a linear complexity w.r.t. the N^2 elements of the resulting vector. Thus, the complexity of all steps in the convolution is $\mathcal{O}(N^2)$ where N is the size of the ETPs being convolved.

Cost of individual operations. In addition to high complexity of convolution, SPTA requires working with very small probabilities. Let us assume a sequence

of 30 instructions described with identical ETPs: $ETP =< (1, 100), (0.2, 0.8) >$. After applying convolutions, the resulting ETP may have latency 30 (i.e. all instructions take 1 cycle) with probability equal to $(0.2)^{30}$. Operating with such small values by using IEEE 754 standard floating-point representations may lead to inaccurate results, even for 64-bit machines.

To overcome the problem of inaccuracy, convolution in SPTA operates on arbitrary-precision floating-point (*apfp*) numbers, whose precision is not limited by fixed-precision arithmetic implemented in hardware. An example of usage of arbitrary-precision arithmetic is public-key cryptography or computation of fundamental mathematical constants such as π to millions of digits.

While *apfp* libraries can provide the required precision, they significantly increase the latency to carry out computations. In that respect, in the canonical form (see Algorithm 1) both floating point numbers in the multiplication in line 5 are *apfp*. This incurs the call to a method of an *apfp* library that results in dozens of assembly instructions to carry out a single *apfp* precision operation. We provide more details on the cost of the *apfp* precision in Section 3.3.

3.2 Optimizing SPTA performance

In this section we analyze a set of optimizations to decrease SPTA timing requirements by either reducing the number of needed convolutions or the cost per convolution: parallelization, sampling and discretization.

3.2.1 Parallelization

We can parallelize convolution in two ways:

- By dividing a single convolution of two ETPs into several threads, referred to as *intra-convolution* parallelism.

- By carrying out several convolutions in parallel, referred to as *inter-convolution* parallelism.

Intra-convolution parallelism.

Canonical convolution (see Algorithm 1) requires iterating through both ETPs being convolved to add their latencies and multiply their probabilities. As there are no data dependencies between computations in each iteration, two nested loops can be easily parallelized in multiple ways.

Let us assume two ETPs containing N and M elements respectively. One of the ways to parallelize convolution is to split one of the vectors - e.g. ETP_1 vector with N elements - into T subETPs of N/T points each, where T is the number of available cores/processors. We convolve each subETP with ETP_2 vector in parallel and get T different ETPs as result. Finally, we insert elements from all obtained ETPs into a single ETP and continue to sorting and normalization steps.

Inter-convolution parallelism.

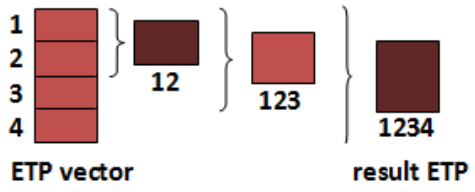
Another way to exploit parallelism is to run convolutions of different ETPs in parallel. In the context of SPTA typically each instruction has its own ETP. The number of instructions per program may be in the order of thousands or hundreds of thousands. Moreover, if ETPs are produced at each instruction invocation (dynamic instructions), the number of ETPs - and so the number of needed convolutions - can be in the order of millions.

To parallelize convolutions of M different ETPs, we split all ETPs into T chunks of $Mc = M/T$ ETPs each (where T is the number of available cores/processors). We assign each chunk to a different core/processor. ETPs inside one chunk can be convolved in two possible ways: sequential order and tree reduction, as illustrated in Figure 3.1.

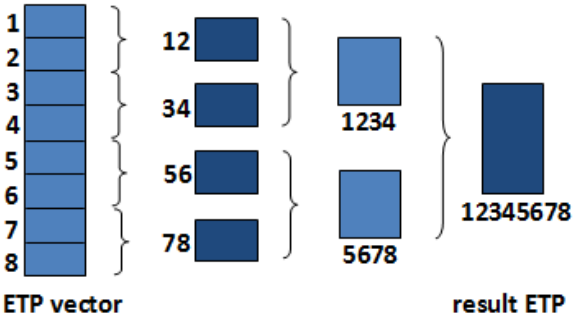
To compare both approaches, we compute the number of required operations (additions of latencies and multiplications of probabilities for each pair of ETPs) to convolve all ETPs inside a chunk. We assume that each ETP contains N elements.

Sequential order within a chunk. We convolve ETPs in-order. The convolution of the first two ETPs requires N^2 operations and results in an ETP with up to N^2 elements. In the next step, we convolve the previously generated ETP with the third ETP in the chunk, which requires up to N^3 operations, and so on. Equation 3.1 shows the maximum number of operations to perform.

$$OpCount_{seq}^{Mc} = \sum_{i=2}^{Mc} \left(N^i\right) \tag{3.1}$$



(a) Sequential order



(b) Tree reduction

FIGURE 3.1: Convolution of ETPs within a chunk

Tree reduction within a chunk. We convolve ETPs in pairs. First we convolve the Mc ETPs, where each convolution requires N^2 operations. Then we convolve the resulting $Mc/2$ ETPs, each containing up to N^2 elements, where each convolution requires up to N^4 operations and so on. Equation 3.2 shows the maximum number of operations to carry out.

$$OpCount_{tree}^{Mc} = \sum_{i=1}^{\lceil \log_2 Mc \rceil} \left(\frac{Mc}{2^i} \times N^{2^i} \right) \tag{3.2}$$

In the case when the number of ETPs is not a power-of-two, we perform an adjustment phase: given M ETPs, we convolve as many pairs as needed so that we obtain M' ETPs where M' is a power-of-two. Formally stated, given M such that $(\log_2 M)\ mod\ 1 \neq 0$, we convolve as many pairs as needed so that we obtain M' where $M > M' > M/2$ and $(\log_2 M')\ mod\ 1 = 0.0$.

3.2.2 Sampling

The number of elements in ETPs increases exponentially with the number of performed convolutions. E.g. convolution of two ETPs of N elements results in an ETP with up to N^2 elements. To keep the number of elements in the ETP under control, and so reduce the number of operations required in further convolutions, sampling techniques are used [27].

In the context of SPTA, sampling consists of two steps:

- Choosing certain number of samples - ⟨latency, probability⟩ pairs - to keep from the original ETP.
- Ensure that the sample ETP is an upper-bound to the original one. This is required to guarantee that probabilistic WCETs are never underestimated. It is achieved by distributing probabilities of omitted points to the right (to the elements with higher latencies).

For example, let us assume the 6-point ETP:

$$ETP^{6p} =< (l_1, l_2, l_3, l_4, l_5, l_6), (0.2, 0.1, 0.05, 0.25, 0.1, 0.3) > \tag{3.3}$$

in which $l_{i+1} > l_i$ and which we want to sample into a 3-point ETP. A possible approach to do so is with the technique called *uniform space re-sampling*, resulting in the ETP shown in Equation 3.4.

$$\begin{aligned} ETP^{3p}_{sampled} &= < (l_2, l_4, l_6), (0.2 + 0.1, 0.05 + 0.25, 0.1 + 0.3) > \\ &= < (l_2, l_4, l_6), (0.3, 0.3, 0.4) > \end{aligned} \tag{3.4}$$

Several ways of sampling an ETP are proposed that ensure that a sample ETP is a reliable upperbound of the original one with low increase in the pessimism [27].

3.2.3 Discretization of probabilities

Under SPTA, the sources of probabilistic behavior usually come from time randomized caches. For instance, in the presence of a data cache as the unique randomization source, loads and stores have the form $< (l_{hit}, l_{miss}), (p_{hit}, p_{miss}) >$, where p_{hit} is the probability of the load/store to hit with its associated latency l_{hit}. The other operations have fixed latency. The basic idea of discretization is to round up (or down) the different probabilities to a multiple of a given rounding value rv, such that all load/stores have only few different probabilities. The outcome is that in many cases we end up convolving N times the same ETP which can be done with a smart implementation of the *power* convolution function. The tradeoff for reduction in execution time is incurred pessimism, by increasing the probabilities of higher latencies.

For instance, let us assume $ETP_1 =< (1, 20), (0.24, 0.76) >$. We perform discretization with a rounding value $rv = 0.1$. To ensure that the new ETP is an upper-bound of the original one, we round up the probability of the higher latency ($l = 20$) to a multiple of 0.1 (from 0.76 to 0.8) and round down the probability of the lower latency ($l = 1$, from 0.24 to 0.2). This results in $ETP_{1rounded} =< (1, 20), (0.2, 0.8) >$. By substracting from the probability of the lower latency the

same value added to the probability of higher latency, the total sum of probabilities in the ETP remains 1.

Formally stated rounding consists in adding ϵ to the probability of the high latency (and subtracting ϵ from the probability of low latency) such that it becomes a multiple of a given rounding value rv, where $rv \leq 1$ and $1 \ mod \ rv = 0$, so that $(p_{high\ lat} + \epsilon) \ mod \ rv = 0$.

The benefit of the rounding step is that different ETPs get an identical form. Assuming $ETP_2 =< (1, 20), (0.22, 0.78) >$, with different probability values than ETP_1, applying discretization will result in a $ETP_{2rounded} =< (1, 20), (0.2, 0.8) >$, identical to $ETP_{1rounded}$. The number of possible forms of ETPs reduces to $g = 1/rv + 1$, where g is typically a relatively low value (e.g., $g = 11$ if $rv = 0.1$). The convolution of ETPs of the same form can be performed by applying fast power operation, explained later in the section. Finally, result ETPs for each form (up to g) are convolved normally, which can be parallelized, as described before in 3.2.1.

Convolution of E copies of the same ETP. The result of convolving E times an ETP is shown in Equation 3.5.

$$ETP^{pow(E)} = ETP_1 \otimes_1 ETP_1 \otimes_2 ETP_1 .... \otimes_{E-1} ETP_1 \tag{3.5}$$

The idea towards reducing the execution time of the power function of convolutions is to breakdown E into an addition of power-of-two values. For example, $E = 7$ can be decomposed into 4, 2 and 1. First, we convolve $ETP_1^{pow(2)} = ETP_1 \otimes ETP_1$. Second, we convolve $ETP_1^{pow(4)} = ETP_1^{pow(2)} \otimes ETP_1^{pow(2)}$. Finally, we convolve at most all those ETPs to get the result ETP, as shown in Equation 3.6. In this case, the power operation requires 4 convolutions, while the sequential approach requires 6.

$$ETP_1^{pow(7)} = ETP_1^{pow(4)} \otimes ETP_1^{pow(2)} \otimes ETP_1^{pow(1)} \tag{3.6}$$

In general, generating the power-of-two ETPs requires $\lceil log_2 E \rceil - 1$ convolutions. In the next step, convolving at most each such ETP (including the original one, ETP_1) requires up to $\lceil log_2 E \rceil - 1$ additional convolutions. In overall, power operation needs to carry out the number of convolutions shown in Equation 3.7, while a sequential approach requires $E - 1$ convolutions.

$$NumConv \leq 2 \times (\lceil log_2 E \rceil - 1) \tag{3.7}$$

Multiple cache memories. To describe discretization in the case of architectures with more randomization sources, let us consider the system with different cache memories (i.e. instruction and data caches). In such case, we independently round miss probabilities up with rv_1 and rv_2 for each cache respectively. Then, we obtain the ETP for the instruction with at most 4 different latencies corresponding to the 4 combinations of hit and miss for both caches. As a result we will have g_1 and g_2 different ETP types for each cache respectively. The resulting number of ETP types for both caches in the first discretization step is, therefore, $g_1 \times g_2$. For instance, if $rv_1 = 0.05$ and $rv_2 = 0.1$, then $g_1 = 21$, $g_2 = 11$ and $g_1 \times g_2 = 231$.

Alternatively, one could compute the ETP for each instruction and each cache independently and then perform the convolution of all those ETPs (2 ETPs per instruction). This is particularly useful if different caches have the same latencies given that this increases the chances of using the power function for ETPs.

Precision. While we use an *apfp* library to gain precision, some optimizations such as discretization and sampling reduce precision. However, those optimizations sacrifice precision in a trustworthy way from a WCET estimation perspective as the resulting ETP always upperbounds the exact one. Conversely, using insufficient precision to operate on probabilities would lead to an uncontrolled loss of precision unacceptable for WCET estimation.

3.3 Experimental Results

This section evaluates the computation time reduction of convolution operation in the SPTA domain achieved by the presented optimizations, when applied in isolation and in a combined manner. In the case of the optimizations which trade off computation time reduction for loss in precision, we also evaluate the increase in pessimism. We integrated all optimizations into an ETP management library, developed in C++.

3.3.1 Experimental conditions

Platform and *apfp* library.

We use a quad-core AMD Opteron^{TM} processor connected to a 32GB DDR2 667 MHz SDRAM. We run a standard Linux distribution on top of it.

For arbitrary-precision floating-point computations we use the GNU *mpfr* (multiple-precision floating-point) library, http://www.mpfr.org/.

The precision of the *mpfr* library was selected according to the criticality level of the target applications. Obviously, the higher the precision the higher each operation takes to execute and the higher are the memory requirements of the library. As an example, for commercial airborne systems at the highest integrity level, called DAL-A, the maximum allowed failure rate per hour of operation [40] in a system component is 10^{-9}. Let us assume that a task is fired every 10^{-2} seconds (i.e. 10^2 activations per second). In order to prevent that task to suffer a timing failure with a probability lower than 10^{-9} per hour, its probability of timing failure per activation, TPF_{act} should be as follows:

$$TPF_{act} \leq \frac{10^{-9}\ timing\ failures/hour}{(3600 \times 10^2\ task\ activations/hour)} \tag{3.8}$$

Therefore, an exceedance probability threshold of 10^{-15} ($TPF_{act} \leq 10^{-15}$) suffices to achieve the highest integrity level. Similarly, exceedance probability thresholds can be derived for other domains and safety levels. We have observed empirically that even if millions of multiplications are performed, a precision of 20 decimal digits suffices to keep accurate results for the 15^{th} decimal digit (and beyond). This means that when enforcing the 20^{th} decimal digit to be rounded up or down for trustworthiness reasons, such pessimism does not propagate up to the 15^{th} decimal digit. Thus, we regard 20 decimal digits as enough for our needs, and select this value as a default value in the experiments. The impact of this parameter in terms of computation cost is studied later in this section.

Optimization parameters.

When applying inter-convolution parallelism, one has to choose between *tree reduction* and *sequential order* when convolving the ETPs within each parallel chunk. Tree reduction typically requires fewer operations than those required with sequential processing ETPs (up to 50% fewer operations). However, it makes ETP size grow faster until their maximum size, which is limited by calling the *sampling* function. Hence with tree reduction most of operations involve working with two ETPs of E elements. Instead, sequential order also make ETPs to grow up to E elements, but keeps convolving it with N-elements ETPs, with $N << E$. This results working with lower-size ETPs and incurring less calls to the sampling function. Overall, this makes the sequential order to work faster than tree reduction and makes it our default choice in the rest of the thesis.

As far as sampling is concerned, many sampling methods have been defined and compared in [27]. Among these, we use *uniform space sampling*, as it provides a good balance among execution time requirements and the pessimism introduced, and is the current state of the art in the field of PTA. In the experiments, unless otherwise stated, sampling will be systematically applied, and the size of ETPs will be limited to 1,024 elements. If larger ETPs are explicitly used (i.e. 2,048 or

ISA					*apfp*				
$\geq$	$=$	$+$	$*$	$/$	$\geq$	$=$	$+$	$*$	$/$
1	1	1	2	3	5	22	17	36	75

FIGURE 3.2: Cost of each operation normalized to native ISA FP add operation

4,096 elements) and sampling is applied, the size of the original ETPs determines the size of the output ETPs.

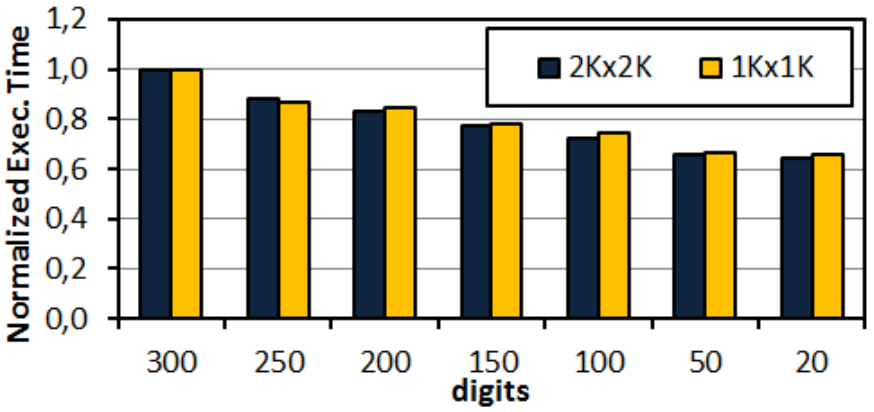
Test-case generation and metrics.

In each experiment we use several ETPs with different number of points. These input ETPs have been generated randomly. To measure the improvement brought by each optimization, we use the execution time reduction, typically w.r.t. non-optimized execution in a single core. Pessimism resulting from some optimizations (sampling and discretization) is also computed w.r.t. to the non-optimized results. Pessimism is measured in terms of weight of the ETP, which is obtained as $W = \sum_{i=1}^{N} p_i \times l_i$ where N is the number of elements in the ETP, and p_i and l_i are the probability and latency at position i respectively [27]. Then, the weight of the ETP after optimizations (W_{optim}) is compared w.r.t. to the ETP without optimizations ($W_{baseline}$).

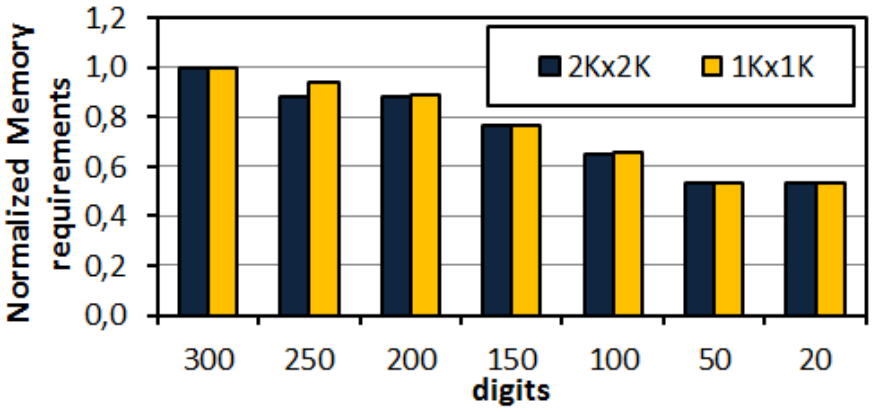
3.3.2 Impact of *apfp* library precision on the cost of each operation

To evaluate the price to pay for having sufficient precision in the ETPs, we first evaluate the execution time of each basic operation used by convolutions (comparison, assignment, addition, multiplication, division). All values are normalized to the execution time of the native FP addition operation, i.e. the operation to add FP numbers in the ISA. Results have been obtained empirically on top of our processor by running micro-benchmarks that exercise the same number of operations of each type.

The results are given in Figure 3.2, with a precision of the *apfp* library set to a high value, here 300 digits. We observe that the impact of the *apfp* library is significant. The comparison is the *apfp* operation with lower overhead being its execution time 5x higher than an ISA regular FP addition. We attribute this to the fact that it is often completed after comparing only a subset of the digits. Addition and assignment have a similar slowdown around 20x while multiplication and division have a latency 36x and 75x higher than the ISA addition respectively. This represents an increment of more than 22x and 26x w.r.t. their ISA counterparts.



(a) Normalized execution time



(b) Normalized memory requirements

FIGURE 3.3: Execution time and memory requirements for different *mpfr* library precisions

To further evaluate the impact of the *apfp* library precision, we run a single-threaded version of the convolution varying the precision of *mpfr* from 300 digits down to 20, which is considered reasonable for SPTA as explained earlier. Figure 3.3 shows the reduction in execution time (3.3(a)) and memory requirements

(3.3(b)) as the number of digits decreases from 300 to 20, when convolving two ETPs. Two sizes of ETPs are experimented: 2,048 (i.e. *2K*) and 1,024 (i.e. *1K*), and sampling is applied. We observe significant reductions of more than 35% and 45% in execution time and memory respectively when moving from 300 to 20 digits, for both ETP sizes.

3.3.3 Parallelization

Intra-ETP parallelization.

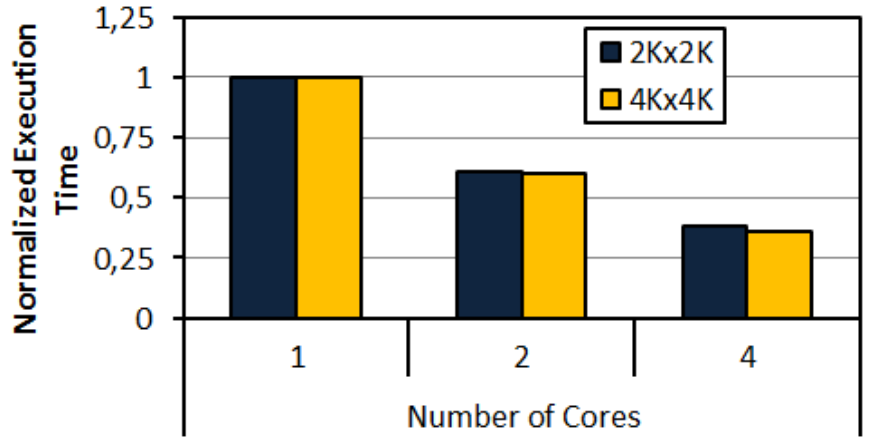
In this experiment we carry out in parallel the convolution of 2 ETPs, with sorting, sampling and normalization turned off. Only the first step of the canonical convolution (see Section 3.1) is executed in parallel and measured. In this way, we obtain an upper-bound of the execution time reduction (scalability) of intra-convolution parallelism. Two different sizes of ETPs are experimented: 2,048 (2K) and 4,096 (4K).

Figure 3.4(a) shows the execution time results when running the convolution on 1, 2 and 4 cores. We observe a good scalability: execution time reduces by 40% with 2 cores and by 65% with 4 cores. The size of the ETPs has a marginal impact.

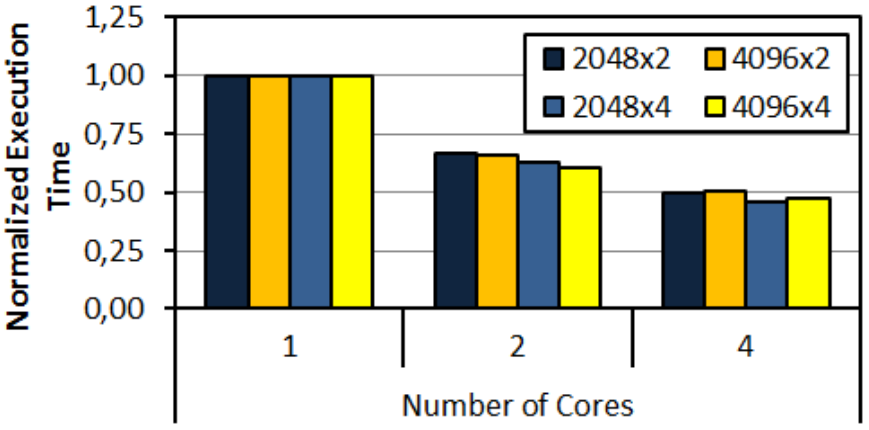
Inter-ETP parallelization.

In contrast to intra-ETP parallelization, inter-ETP parallelization does not parallelize one convolution, but instead splits a sequence of convolutions into chunks to be processed in parallel. In this experiment, given a vector of M ETPs to convolve, we measure the benefit of dividing it into $T \in [1, 4]$ chunks, each of which is processed in parallel on one core. The ETPs in each chunk are processed in sequential order.

Figure 3.4(b) shows the execution time benefit of inter-ETP parallelization when convolving vectors of 2,048 and 4,096 ETPs. Results are also shown across different



(a) E.T. of intra-ETP parallelization



(b) E.T. of inter-ETP parallelization

FIGURE 3.4: Impact of parallelization on execution time

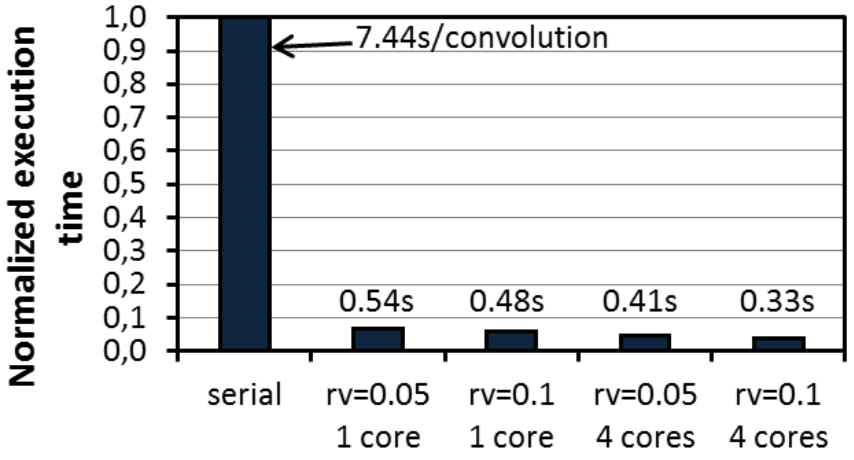
numbers of elements per ETP, namely, 2 and 4. Results do not reach optimal scaling due to: (i) the intrinsic overhead of parallelization (e.g., spawning and synchronizing threads) and (ii) because eventually the number of ETPs to convolve is lower than the core count, thus leaving some cores idle.

3.3.4 Probability discretization

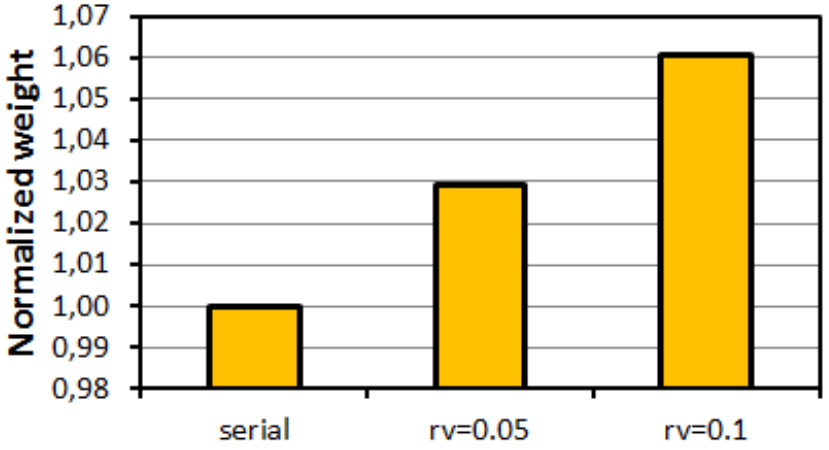
In this experiment, we assess the execution time benefits and impact on pessimism introduced by probability discretization. For this experiment we carry out the convolution of a vector of 4,096 ETPs of 2 elements each[1]. Those ETPs are

[1]A two-point ETP represents an architecture with a single level of cache, e.g. the instruction cache, where each ETP takes the form: $< (l_{hit}, l_{miss}), (p_{hit}, p_{miss}) >$

randomly generated. We carry out the evaluation for two different rv values: 0.05 and 0.1.



(a) Run time of discretization



(b) Pessimism introduced

FIGURE 3.5: Evaluation of the *Discretization* optimization

Figure 3.5 shows the results, obtained by averaging the ETP weight and execution times on 1000 runs. When run on single core (three leftmost bars of Figure 3.5(a)), we observe that with $rv = 0.05$, we obtain an execution time reduction of more than 80%. With $rv = 0.1$ there is an extra slight reduction in the execution time. However, in terms of pessimism (ETP weight, shown in Figure 3.5(b)), $rv = 0.05$ shows to have low pessimism. The increase in pessimism of $rv = 0.1$ does not pay off its extra small reduction in execution time.

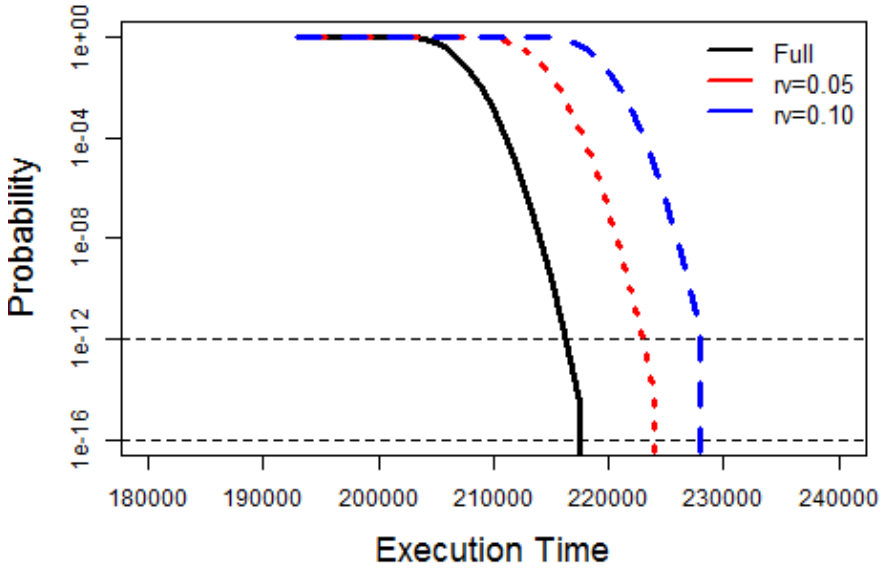


FIGURE 3.6: pWCET estimates with and without discretization

Figure 3.6 compares the pWCET estimates obtained after convolving 4,096 random ETPs (with 2 elements each) when discretization is not applied, and when it is applied with $rv = 0.05$ and $rv = 0.1$. We observe that with discretization obtained pWCET estimates are more pessimistic than when not using discretization. However, the pessimism introduced is relatively small. For instance, for a cutoff probability of 10^{-12} the overestimation is 3.1% for $rv = 0.05$ and 5.5% for $rv = 0.1$.

3.3.5 Combination of techniques

The two rightmost bars in Figure 3.5(a) show the result of combining discretization and hybrid (inter and intra) parallelization. We observe that the combination of both reduces the cost of convolutions to less than 5% of the cost of the non-optimized convolution method, thus showing that benefits of optimizations are geometric. In terms of absolute execution time, the cost of one convolution reduces from 7.44s down to 0.33s. Thus, if a program has 100,000 instructions, those optimizations reduce convolution cost from 8.6 days down to 9.2 hours. While such cost is non-negligible yet, we regard it as affordable and it can be further reduced if other optimizations like fast-fourier transformation are applied.

So far we have focused on the convolution which handles sequential sections of instructions. In the presence of control flow constructs (CFC) such as `if-then`, `if-then-else` or `switch`, another operation is needed, which is called *envelope* [41] or maximum [3]. In the presence of those CFC, convolution needs to be applied inside each branch of the CFC to obtain an ETP for each of them. Then, the envelope operation computes an ETP upper-bounding all of them so that for any latency the accumulated probability of the envelope ETP is equal or higher than for any of the input ETPs. This way the set of ETPs of the different branches can be replaced by the envelope one. By doing this CFCs can be removed and convolution can be applied normally using the envelope ETP at the expense of some pessimism. The cost of the envelope operation has been proven low (linear with the length of the input ETPs) [3, 41] and it only impacts in which order convolutions can be carried out. However, the implementation and evaluation of further convolution optimizations is left as part of our future work.

Chapter 4

ACET in Measurement-Based Probabilistic Timing Analysis

4.1 Probabilistic analytic cache modeling (PACO)

In this chapter we describe and evaluate PACO, the method for quick estimation of average performance of time randomized caches. The method uses a set of approximation formulas for probabilities of miss in TR caches, which we present next for variety of cache organizations.

We analyze copy-back (CB) and write-through (WT) caches, and 3 different configurations for the associativity: (FA) fully-associative, (DM) direct-mapped, and (SA) set-associative (4-way in the evaluation section). Thus, there are 6 different cache types: (CB-FA), (CB-DM), (CB-SA), (WT-FA), (WT-DM), (WT-SA). We consider split data (DL1) and instruction (IL1) first level caches and unified second level cache (UL2).

We start our analysis with fully-associative and direct-mapped caches in which only the random replacement and the random placement policies are respectively used. Finally, we focus on set-associative caches that deploy both random placement and replacement. We use the same reference sequence ($A_{j-1}, B_1^1, B_2^2, C_1^3 ..., F_1^k, A_j$) and the same nomenclature as in the Section 2.3.3.

4.1.1 Copy-back Fully-associative Caches (CB-FA)

$\mathbf{P_{miss}}$ for DL1 and IL1 are called P^{DL1}_{miss} and P^{IL1}_{miss} respectively. For a copy-back setup P_{miss} is as shown in Equation 2.2, in which P_{miss} for a given access depends on the number of accesses, and their associated probability, between A_j and the previous access to the same line A_{j-1}.

In the case of the DL1 only memory operations (mop^l), i.e. loads and stores, access the DL1. Hence, for the DL1, mop^l in Equation 2.2 represents all memory operations between A_{j-1} and A_j, and so we have Equation 4.1 where W_{DL1} is the number of ways in the DL1.

$$P^{DL1}_{miss_{A_j}}(W_{DL1}) = 1 - \left(\frac{W_{DL1}-1}{W_{DL1}}\right)^{\sum_{l=1}^{l=k} P^{DL1}_{miss_{mop^l}}} \tag{4.1}$$

For the IL1 X^l stands for all instructions between A_{j-1} and A_j, i.e. $inst^l$.

$$P^{IL1}_{miss_{A_j}}(W_{IL1}) = 1 - \left(\frac{W_{IL1}-1}{W_{IL1}}\right)^{\sum_{l=1}^{l=k} P^{IL1}_{miss_{inst^l}}} \tag{4.2}$$

On every miss of an access X^l between two accesses to the same line, A_{j-1} and A_j, a random eviction is carried out. On every eviction the probability of not evicting A_j is $(W_{DL1}-1)/W_{DL1}$ for DL1 (IL1 is analogous). The exponent in Equation 4.1 accumulates the miss probability of all accesses between A_{j-1} and A_j. This formula approximates P_{miss} based on the expected number of evictions produced by all accesses occurred since the previous access to A. Note that for the first access A_1 we have that $P_{miss_{A_1}} = 1$.

$\mathbf{P_{miss}}$ for UL2 considers the number of evictions produced between A_{j-1} and A_j, since it determines the number of misses in UL2: NM_{UL2}. A data access misses in the UL2 if it misses in the DL1 first, which occurs with probability $P^{DL1}_{miss_{X^l}}$ and it also misses in the UL2, which occurs with probability $P^{dUL2}_{miss_{X^l}}$. Both probabilities are computed as shown in 4.1. NM_{UL2} is also affected by the number of misses in the IL1 that also miss in UL2.

$$NM_{UL2} = \sum_{l=1}^{l=k} \left[\left(P^{DL1}_{miss_{X^l}} \times P^{dUL2}_{miss_{X^l}} \right) + \left(P^{IL1}_{miss_{X^l}} \times P^{iUL2}_{miss_{X^l}} \right) \right]$$

Overall the UL2 miss probability for A_j is given by:

$$P^{UL2}_{miss_{A_j}}(W_{UL2}) \;=\; 1 - \left(\frac{W_{UL2} - 1}{W_{UL2}} \right)^{NM_{UL2}} \tag{4.3}$$

Note that one source of inaccuracy for this approximation is the fact that, in pipelined processors, instruction and data accesses are not aligned because a data access can suffer evictions from younger instruction accesses (so accesses after A_j) that reach UL2 early in the pipeline, and because an instruction access can suffer evictions from older data accesses (so accesses before A_{j-1}) that reach UL2 later in the pipeline. For instance, in the sequence $(B_1 A_1 B_2 B_3 A_2 B_4)$ a data access of A_2 could suffer an eviction from the instruction access of B_4 and an instruction access of A_1 could be evicted by a data access of B_1.

4.1.2 Copy-back Direct-Mapped Caches (CB-DM)

$\mathbf{P_{miss}}$ for DL1 and IL1. While any given cache line A can be evicted by any new line fetched from memory in a fully-associative cache, only lines placed in the same set as A can evict it. Indeed, any such line will evict A in a direct-mapped cache. Random placement leads to a probability of $\frac{1}{S}$ of two cache lines to be placed in the same set given S cache sets. Thus, given the same access sequence as before, $(A_{j-1}, X^1, ..., X^k, A_j)$, where A_{j-1} and A_j correspond to accesses to the same cache line, and no X^l (where $1 \leq l \leq k$) accesses the same cache line as A_j, the probability of A_j to miss in cache is as follows:

$$P^{xL1}_{miss_{A_j}}(S) = 1 - \left(\frac{S - 1}{S} \right)^q \tag{4.4}$$

Where q stands for the number of unique (i.e. non-repeated) cache lines among all X^l. Repeated addresses are disregarded because they access always the same set so either all of them cannot evict A or all of them would evict it [21].

$\mathbf{P_{miss}}$ for UL2. The equation above is valid for DL1 and IL1, while for the UL2, P_{miss} is approximated as follows:

$$P^{UL2}_{miss_{A_j}}(S_{UL2}) = P^{xL1}_{miss_{A_j}}(S_{xL1}) \times \left(1 - \left(\frac{S_{UL2}-1}{S_{UL2}}\right)^{q_I+q_D}\right) \tag{4.5}$$

where q_I and q_D are the number of unique instruction and data addresses respectively accessed by all instructions in between the one accessing A_{j-1} and the one accessing A_j.

4.1.3 Copy-back Set-associative Caches (CB-SA)

$\mathbf{P_{miss}}$ for DL1 and IL1. P_{miss} values in direct-mapped and fully-associative caches are independent given that P_{miss} depends on unique addresses in the former and on previous P_{miss} values in the latter. As a result, probability of both events to occur can be obtained by multiplying their respective probabilities [21].

$$P_{miss_{A_j}}(W,S) = \left(1 - \left(\frac{W-1}{W}\right)^{\sum_{l=1}^{l=k} P_{miss_{X^l}}}\right) \times \left(1 - \left(\frac{S-1}{S}\right)^{q}\right) \tag{4.6}$$

Equation 4.6 is the product of equations 2.2 and 4.4, meaning that an access is a miss in cache if any X^l accessed the same set (second part of the equation) and it randomly evicted A in that set (first part of the equation).

In this equation we identify a source of inaccuracy due to the fact that the first part considers *all* evictions occurred in between A_{j-1} and A_j when, instead, it should only consider those occurring in the same cache set. Therefore, as random placement is intended to distribute randomly and evenly addresses across the

different cache sets, we should divide by S, the number of sets, the exponent of the first part:

$$P_{miss_{A_j}}(W,S) = \left(1-\left(\frac{W-1}{W}\right)^{\frac{\sum_{l=1}^{l=k} P_{miss}{}_{X^l}}{S}}\right) \times \left(1-\left(\frac{S-1}{S}\right)^{q}\right) \tag{4.7}$$

$\mathbf{P_{miss}}$ for UL2. In the case of UL2, P_{miss} for access A_j in our reference sequence is as follows, where $xL1$ represents the cache accessed by A_j, that is, $IL1$ or $DL1$:

$$P^{UL2}_{miss_{A_j}}(W_{UL2},S_{UL2}) = P^{xL1}_{miss_{A_j}}(W_{xL1},S_{xL1}) \times P^{UL2-only}_{miss_{A_j}} \tag{4.8}$$

$P^{UL2-only}_{miss_{A_j}}$ is the miss probability for A_j as if it accessed UL2 directly (omitting xL1):

$$P^{UL2-only}_{miss_{A_j}}(W_{UL2},S_{UL2}) = \left(1-\left(\frac{W_{UL2}-1}{W_{UL2}}\right)^{\frac{\sum_{l=1}^{l=k} P_{miss}{}_{X^l}}{S_{UL2}}}\right) \times \left(1-\left(\frac{S_{UL2}-1}{S_{UL2}}\right)^{q_I+q_D}\right) \tag{4.9}$$

4.1.4 Write-through Caches (WTx)

The case of write-through caches is analogous to that of copy-back ones with the following differences:

- P^{DL1}_{miss} for DL1 accesses of store instructions is irrelevant from a performance perspective as those accesses are forwarded to UL2 anyway.
- As we assume that UL2 is always copy-back, P^{UL2}_{miss} must consider those accesses caused by IL1 misses, DL1 load misses and *all* DL1 store accesses (regardless of whether they hit or miss).

Other than that, those approximations used for copy-back caches remain valid for write-through ones.

4.1.5 Multiple Addresses per Cache Line

When the addressable unit is smaller than a cache line, accesses to different addresses can be mapped to the same cache line. This has no impact on our previous formulation. For instance, let us assume the sequence ($A_{j-1}, B_1^1, C_1^2, D_1^3, E_1^4$..., F_1^k, A_j), in which B and C go to the same line.

We can simply abstract this sequence as ($A_{j-1}, B_1^1, B_2^2, D_1^3, E_1^4$..., F_1^k, A_j), hence considering that the access to C corresponds to another access to B. This allows us applying the same formulation as above to compute P_{miss}.

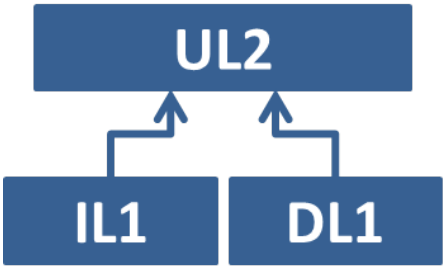
4.2 Experimental Results

This section evaluates the accuracy of PACO to estimate P_{miss} (and so P_{hit}) probabilities. For that purpose, we compare PACO against simulation where 100,000 simulations are used to obtain figures highly accurate for the leftmost decimal digits of the different probabilities.

We consider two cache setups, *1-level* and *2-level*.

– Under 1-level only the first level instruction (IL1) and data (DL1) TR caches are used. In this setup DL1 is copy back and the IL1 is read-only.

– 2-level also includes a unified second level (UL2) TR cache which is accessed in case of miss in IL1 or DL1, see Figure 4.1. This is the most complex hierarchy shown in [22] and conclusions can be extrapolated to larger hierarchies with third or even fourth level caches. In this setup, IL1 is read-only, DL1 is write-through and UL2 is copy-back. DL1 is no-write-allocate, so store misses do not fetch new data to DL1. All store instructions reach UL2 regardless of whether they hit in DL1.



	IL1	DL1	UL2
1 level	Read only	Copy Back	-
2 levels	Read only	Write Through	Copy Back

FIGURE 4.1: Cache hierarchy and setups considered.

In our reference cache setup, DL1 and IL1 are 8KB in size and have 32 bytes/line. UL2 is 64KB and has 32 bytes/line. We consider direct-mapped, fully-associative and 4-way set-associative caches. Caches are non-inclusive, hence imposing no constraint on whether contents in IL1 or DL1 must or must not be in UL2. Differences in the behavior w.r.t. other inclusion policies have been shown to be rather small [22].

The evaluation has been conducted on the EEMBC Autobench benchmark suite [42], which is a well-known suite reflecting the current real-world demand of some automotive embedded systems. Address traces for PACO and simulation measurements have been collected using the reference input provided together with the benchmark suite. If the analysis needs to be performed for multiple input sets, such analysis can be performed individually for each input set and combined analogously as for the case of running simulations.

Results are reported in terms of the following figures:

- *Per-access evaluation.* For each access in the program we compute the *absolute* difference between the probabilities provided by PACO and those obtained through simulation. For instance if $P_{miss}^{sim} = 10.5\%$ and $P_{miss}^{PACO} = 11.5\%$ the difference is 1% [1]. We then obtain the average and standard deviation of those values across each benchmark for each one of the caches (DL1, IL1 and UL2) in all those 6 setups described in Section 4.1. Per-benchmark results are averaged thus giving each benchmark the same weight.

[1] This can be expresed in *percentage points* (pp). A *pp* is the unit for the arithmetic difference of two percentages. e.g. going from 1% to 9% is an 8 percentage point increase. For the sake of simplicity we do not use percentage points.

- *Per-program evaluation.* Users may be interested in analyzing probabilities at a much coarser granularity than per-access. Therefore, it may be interesting estimating average probabilities for a given program. Thus, we also report per-program results as follows. We compute the actual difference *(not absolute)* between the probabilities provided by PACO and those obtained through simulation. We average those values across each benchmark for each one of the caches in all setups. This provides the actual inaccuracy per program for each cache in each cache organization. Then, we compute the absolute values for each program and report the average difference for each cache in each scenario.

Let us introduce a simple example to illustrate the difference between per-access and per-program results. Let us assume a single cache and 4 accesses whose P_{miss} through simulation is 0.2, 0.1, 0.1, 0.3 respectively and 0.25, 0.1, 0.2, 0.25 through PACO. In this case, per-access average P_{miss} error is 0.05 $\left(\frac{0.05+0+0.1+0.05}{4}\right)$ whereas per-program accuracy error is 0.025 $\left(\frac{0.05+0+0.1-0.05}{4}\right)$. As shown, per-program error can only be lower because errors can cancel out. In the example, underestimation for the 4^{th} access partially offsets the overestimation for the 1^{st} and 3^{rd} accesses.

Finally, we also report results in terms of computational cost. We compare the implementation of PACO w.r.t. the simple cache simulator implemented as baseline. Both of them have been coded from scratch following usual programming guidelines, compiled analogously and no specific code optimization has been applied. Execution times have been obtained on top of a Xeon Dual-Core 5148 operating at 2.33GHz with 12 GB of DRAM.

4.2.1 Per-access Results

As shown in Table 4.1, P_{miss} estimates obtained with PACO are highly accurate for all fully-associative (FA) and direct-mapped (DM) setups with an average

TABLE 4.1: Per-access P_{miss} accuracy. (Avg stands for average and Std for standard deviation.)

Cache setup	**DL1**		**IL1**		**UL2**	
	Avg	**Std**	**Avg**	**Std**	**Avg**	**Std**
CB-FA	0.02%	0.04%	0.02%	0.04%	N/A	N/A
CB-DM	0.02%	0.03%	0.07%	0.07%	N/A	N/A
CB-SA	1.02%	1.74%	2.59%	3.02%	N/A	N/A
WT-FA	0.01%	0.02%	0.02%	0.04%	0.26%	1.33%
WT-DM	0.01%	0.03%	0.07%	0.07%	0.93%	1.31%
WT-SA	0.54%	1.28%	2.59%	3.02%	2.33%	5.11%

TABLE 4.2: Per-program P_{miss} accuracy.

Cache setup	**DL1**	**IL1**	**UL2**
	Average	**Average**	**Average**
CB-FA	0.00%	0.00%	N/A
CB-DM	0.00%	0.02%	N/A
CB-SA	0.68%	2.49%	N/A
WT-FA	0.00%	0.00%	0.17%
WT-DM	0.00%	0.02%	0.88%
WT-SA	0.36%	2.49%	2.21%

TABLE 4.3: Absolute P_{miss} values.

Cache setup	**DL1**	**IL1**	**UL2**
	Average	**Average**	**Average**
CB-FA	2.04%	0.95%	N/A
CB-DM	7.85%	1.66%	N/A
CB-SA	2.06%	1.05%	N/A
WT-FA	10.97%	0.95%	13.86%
WT-DM	12.63%	1.66%	12.01%
WT-SA	10.90%	1.05%	12.42%

difference of 0.03% for DL1 and IL1 caches implementing copy-back (CB) or write-through (WT) policies. Results for DL1 and IL1 still offer good accuracy for set-associative (SA) caches although less than for FA and DM setups, thus indicating that there is potential for improvement of the model. Results for the UL2 cache are less accurate as they accumulate P_{miss} inaccuracies in DL1/IL1 caches determining how many UL2 accesses occur, on top of the UL2 cache inaccuracy itself. Note that IL1 results for CB and WT policies are identical as all IL1 accesses are read accesses, and so the write policy has no effect.

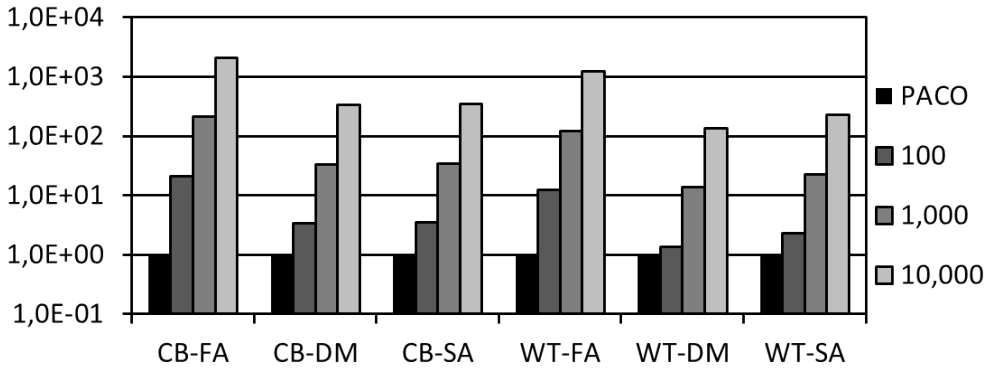


FIGURE 4.2: Execution time of simulations normalized w.r.t. PACO.

4.2.2 Per-program Results

As shown in Table 4.2, average P_{miss} estimates obtained with PACO for full programs are more accurate than per-access ones as inaccuracies cancel out to some degree. Again, accuracy for DM and FA caches is much higher than for SA ones. In fact, results for DM and FA DL1 and IL1 caches only show negligible inaccuracy.

For the sake of reference, absolute P_{miss} values are shown in Table 4.3, thus illustrating that relative inaccuracies are low except for some SA caches.

4.2.3 Execution Time Cost

We have also compared the cost of executing our model w.r.t. the cost of simulating cache behavior, which we regard as the only alternative to obtain results at the same granularity as PACO, so per-access and per-cache memory for any target cache setup. Results are shown in Figure 4.2. As shown, PACO has a cost similar to that of running 31 simulations on average (between 4 and 74 simulations for different setups), so always lower than that of performing 100 simulations. The relative cost of simulations grows exponentially with the number of simulations. We observe that such trend holds across configurations. Furthermore, the relative cost of PACO for DM and SA caches, the ones with higher cost, can be further reduced if multiple cache setups need to be evaluated. This occurs because most computation time of PACO is spent computing the unique address reuse distance (q in Equation 4.4), which needs to be computed only once regardless of the number

of cache setups to be evaluated. Conversely, the cost of simulation grows linearly with the number of cache setups. For instance, if we evaluated 100 different cache setups, the average cost of PACO would be as low as that of 4 simulations per setup, thus 25 times lower than using 100 simulations per setup.

In summary, our per-access and per-program results show that PACO accuracy error is within 0.7% of that obtained with 100,000 simulations for P_{miss} on average. Thus, PACO provides high accuracy for P_{miss} with low execution time requirements.

Chapter 5

Conclusions and Future work

5.1 Conclusions

PTA has been regarded as a powerful approach to obtain reliable and tight WCET estimates.

The static variant of PTA, SPTA, requires the use of convolutions, whose computational cost is high. In this thesis we have identified some features of convolutions that require a large number of computations and implemented a set of optimizations to reduce their cost. Those optimizations, integrated into a software library, include precision-preserving optimizations (e.g., parallelization), as well as optimizations that trade off some accuracy for some computational cost reduction while preserving reliability. Among those, discretization shows to be the most effective solution. Our results prove the effectiveness of the different optimizations and a small subset of them show a combined execution time reduction down to less than 5% of that of the non-optimized version.

The measurement-based variant of PTA, MBPTA, have been deeply studied from a WCET perspective, but there is a lack of efficient ways to estimate ACET. Time randomized caches are the resources with highest impact on average performance, due to the fact that having a hit or miss in a cache leads to huge differences in execution time. So far estimating the cache hit/miss probabilities has been

done with a large number of simulations. In this thesis we introduce PACO to efficiently estimate hit and miss probabilities for a wide variety of cache setups and organizations. Our results show that PACO obtains an accuracy within 2.6% across different setups and caches with low computational cost.

5.2 Future work

On the SPTA part, another approach to speed-up convolutions is to use Fourier Transformation, and in particular its discrete fast version (DFT). This approach needs first to convert the distribution from the time domain to the frequency domain using DFT. Then, according to the convolution theorem, a point-wise multiplication is applied, which is equivalent to the convolution in the time domain. Finally, inverse DFT is performed to obtain the distribution in the time domain. As part of our future work we plan to evaluate the use of DFT to speed up convolutions as well as to explore further optimizations.

On the MBPTA part, as part of our future work we plan to extend our evaluation of hit/miss approximations formulas to a wider variety of cache setups, find more accurate approximations for set-associative caches and extend our model to approximate the probability of evicting dirty lines. We also plan to optimize the implementation of PACO for a further execution time cost reduction.

Chapter 6

Published work

- Suzana Milutinovic, Jaume Abella, Damien Hardy, Eduardo Quiñones, Isabelle Puaut, Francisco J. Cazorla, *"Speeding up Static Probabilistic Timing Analysis"*, in proceedings of the 28th GI/ITG International Conference on Architecture of Computing Systems (ARCS), Porto (Portugal), March 24-27 2015.

- Suzana Milutinovic, Eduardo Quiñones, Jaume Abella, Francisco J. Cazorla, *"PACO: Fast Average-Performance Estimation for Time-Randomized Caches"*, in proceedings of the 52nd ACM/IEEE Design Automation Conference (DAC), San Francisco (California), June 7-11 2015.

Bibliography

[1] Paul Lokuciejewski and Peter Marwedel. *Worst-Case Execution Time Aware Compilation Techniques for Real-Time Systems.* Springer, 2011.

[2] R. Wilhelm et al. The worst-case execution time problem: overview of methods and survey of tools. *Trans. on Embedded Comp. Systems*, 7(3):1–53, 2008.

[3] G. Bernat, A. Colin, and S.M. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS*, 2002.

[4] J. Hansen, S. Hissam, and G. A. Moreno. Statistical-based WCET estimation and validation. In *WCET Workshop*, 2009.

[5] F. J. Cazorla et al. PROARTIS: Probabilistically analyzable real-time systems. *ACM Trans. on Embedded Computing Systems*, 12(2s), 2013.

[6] L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.

[7] F. Wartel et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.

[8] R.I. Davis et al. Analysis of probabilistic cache related pre-emption delays. In *ECRTS*, 2013.

[9] S. Altmeyer and R. I. Davis. On the correctness, optimality and precision of static probabilistic timing analysis. In *DATE*, 2014.

[10] J. Abella et al. On the comparison of deterministic and probabilistic WCET estimation techniques. In *ECRTS*, 2014.

[11] S. Kotz and S. Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000. ISBN 1860942245, 9781860942242.

[12] L. Kosmidis et al. Containing timing-related certification cost in automotive systems deploying complex hardware. In *DAC*, 2014.

[13] L. Kosmidis, E. Quinones, J. Abella, T. Vardanega, I. Broster, and F.J. Cazorla. Measurement-based probabilistic timing analysis and its impact on processor architecture. In *17th Euromicro Conference on Digital System Design (DSD), 2014*, pages 401–410, 2014.

[14] F. Mueller. Predicting instruction cache behavior. *Language, Compilers and Tools for Real-Time Systems*, 1994.

[15] C. Ferdinand and R. Wilhelm. Fast and Efficient Cache Behavior Prediction for Real-Time Systems. *Real-Time System Journal*, XVII:131–181, 1999.

[16] F. Mueller. Timing analysis for instruction caches. *Real-Time Systems Journal - Special issue on worst-case execution-time analysis*, 2000.

[17] C. Ferdinand et al. Reliable and precise WCET determination for a real-life processor. In *EMSOFT*, 2001.

[18] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. In *WCET Workshop*, 2009.

[19] D. Hardy and I. Puaut. WCET analysis of multi-level non-inclusive set-associative instruction caches. In *RTSS*, 2008.

[20] G. Bernat, A. Colin, and J. Esteves. Considerations on the LEON cache effects on the timing analysis of on-board applications. In *DASIA*, 2007.

[21] L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.

[22] L. Kosmidis et al. Multi-level unified caches for probabilistically time analysable real-time systems. In *RTSS*, 2013.

[23] J. Reineke. Randomized caches considered harmful in hard real-time systems. *LITES*, 1(1), 2014.

[24] M. Patte and V. Lefftz. System impact of distributed multi core systems. Technical Report ESTEC Contract 4200023100, European Space Agency, 2011.

[25] C.J. Turner, V.C. Bhavsar, and P.R. Pochec. Parallel implementations of convolution and moments algorithms on a multi-transputer system. *Microprocessors and Microsystems*, 19(5), 1995.

[26] H.-M. Yip, I. Ahmad, and T.-C. Pong. An efficient parallel algorithm for computing the gaussian convolution of multi-dimensional image data. *J. Supercomput.*, 14(3), 1999.

[27] D. Maxim, M. Houston, L. Santinelli, G. Bernat, R.I. Davis, and L. Cucu. Re-sampling for statistical timing analysis of real-time systems. In *RTNS*, 2012.

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

[29] M.K. Gardner and J.W. Lui. Analyzing stochastic fixed-priority real-time systems. In *the 5th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS99)*, pages 44–58, 1999.

[30] L. Abeni and G. Buttazzo. Integrating multimedia applications in hard real-time systems. In *Proceedings of Real-Time Systems Symposium (RTSS)*, pages 4–13, 1998.

[31] T.S. Tia, Z. Deng, M. Shankar, M. Storch, J. Sun, L.C. Wu, and J.S Liu. Probabilistic performance guarantee for real-time tasks with varying computation times. In *the 2nd IEEE Real-Time and Embedded Technology and Applications Symposium (RTAS95)*, pages 164–174, 1995.

[32] Dorin Maxim, Olivier Buffet, Luca Santinelli, Liliana Cucu-Grosjean, and Robert I. Davis. Optimal priority assignment algorithms for probabilistic real-time systems. In *19th International Conference on Real-Time and Network Systems, (RTNS)*, pages 129–138, 2011.

[33] J.P. Lehoczky. Real-time queueing theory. In *Proceedings of Real-Time Systems Symposium (RTSS)*, pages 186–195, 1996.

[34] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *Proceedings of Real-Time Systems Symposium (RTSS)*, 2001.

[35] David Griffin and Alan Burns. Realism in Statistical Analysis of Worst Case Execution Times. In *Proceedings of International Workshop on Worst-Case Execution Time Analysis (WCET)*, pages 44–53, 2010.

[36] 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 *24th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 91–101, 2012.

[37] Petar Radojković, Vladimir Čakarević, Miquel Moretó, Javier Verdú, Alex Pajuelo, Francisco J. Cazorla, Mario Nemirovsky, and Mateo Valero. Optimal task assignment in multithreaded processors: A statistical approach. *SIGARCH Comput. Archit. News*, 40(1):235–248, March 2012.

[38] Francisco J. Cazorla, Tullio Vardanega, Eduardo Quiñones, and Jaume Abella. Upper-bounding Program Execution Time with Extreme Value Theory. In *13th International Workshop on Worst-Case Execution Time Analysis*, volume 30 of *OpenAccess Series in Informatics (OASIcs)*, pages 64–76, 2013.

[39] S. Zhou. An efficient simulation algorithm for cache of random replacement policy. In *NPC*, 2010.

[40] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.

[41] L. Cucu, F.J. Cazorla, J. Abella, and M. Houston. D3.4 probabilistic and statistical techniques for timing analysis in single core. Technical report, PROARTIS, 2013. URL http://www.proartis-project.eu/system/files/D3.4%20Probabilistic%20and%20Statistical%20Techniques%20for%20Timing.pdf.

[42] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.