D3.4 Probabilistic and Statistical Techniques for Timing Analysis in Single Core
Version 1.0

Document Information

<table>
<thead>
<tr>
<th>Contract Number</th>
<th>249100</th>
</tr>
</thead>
<tbody>
<tr>
<td>Project Website</td>
<td><a href="http://www.proartis-project.eu">www.proartis-project.eu</a></td>
</tr>
<tr>
<td>Contractual Deadline</td>
<td>m27</td>
</tr>
<tr>
<td>Dissemination Level</td>
<td>D1.2 Restricted*/Public¹</td>
</tr>
<tr>
<td>Nature</td>
<td>Report</td>
</tr>
<tr>
<td>Lead Authors</td>
<td>Liliana Cucu, Francisco J. Cazorla, Jaume Abella and Michael Houston</td>
</tr>
<tr>
<td>Contributors</td>
<td>All members of all institutions (BSC, RAPITA, UNIPD, INRIA, AFS)</td>
</tr>
<tr>
<td>Reviewers</td>
<td>Francisco J. Cazorla</td>
</tr>
<tr>
<td>Keywords</td>
<td>Probabilistic Timing Analysis, Single Core, Time Composability, pWCET estimations</td>
</tr>
</tbody>
</table>

Notices:

The research leading to these results has received funding from the European Community’s Seventh Framework Programme ([FP7/2007-2013] under grant agreement n° 249100.

©2010 PROARTIS Consortium Partners. All rights reserved.

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

<table>
<thead>
<tr>
<th>Version</th>
<th>Description of change</th>
</tr>
</thead>
<tbody>
<tr>
<td>v1.0</td>
<td>Initial Draft released to the European Commission</td>
</tr>
</tbody>
</table>
## Contents

1 Introduction ........................................ 5

2 Probabilistic Time Analysis Techniques ............ 7
   2.1 Main notations and definitions .................. 7
   2.2 SPTA ........................................ 8
      2.2.1 Reducing the complexity of SPTA ........... 10
   2.3 MBPTA ....................................... 13
      2.3.1 Steps in the application of EVT ............. 15
      2.3.2 EVT for single-path WCET .................. 16
      2.3.3 Multi-path programs ....................... 19
   2.4 HyPTA: Rapita Demonstrator .................... 20

3 Time Composability aspects of PROARTIS ........... 23
   3.1 Time Composability in PROARTIS ............... 23
      3.1.1 Assumptions ................................ 24
      3.1.2 SPTA ..................................... 24
      3.1.3 MBPTA .................................... 27

4 Results for standard benchmarks and Rapita Demonstrator ........ 31
   4.1 Hardware Experimental Setup .................... 32
      4.1.1 Common processor core architecture ........ 32
      4.1.2 Cache architecture ........................ 33
   4.2 Benchmark Suites ................................ 35
   4.3 Icache-only Setup: Results ..................... 36
      4.3.1 MBPTA-EVT autoevaluation ................. 36
   4.4 Single-level cache setup: Results ............... 42
      4.4.1 Single-Path Programs ....................... 42
      4.4.2 Multiple-Path Programs ..................... 43
      4.4.3 Cache configuration sensitivity ............. 44
   4.5 Multiple-level cache setup: Results ............ 45
      4.5.1 Single-path programs ....................... 45
      4.5.2 multi-path programs ....................... 47
   4.6 Comparison with Static Timing Analysis and deterministic architectures .................... 49
   4.7 Probabilistic execution time bounds under different execution conditions ................... 52
   4.8 HyPTA: Rapita Systems Demonstrator .......... 55
      4.8.1 Results of HyPTA on the missile guidance simulator ........ 56
4.9 Software-based Randomisation ........................................ 57
Acronyms and Abbreviations ............................................. 61
1

Introduction

This section presents three main parts:
- The Probabilistic Timing Analysis (PTA) techniques proposed for the single core case that are built on top of the properties provided by the PROARTIS platform (D1.2).
- The Time Composability aspects emanating from our probabilistic approach to timing analysis.
- The evaluation results we have obtained during this second phase of the project other than the results with the industrial cases studies, which are presented in D4.3. Note that this set of results summarise the effort described in D1.2 and D3.4 itself since separate results cannot be provided.

The objective of PTA is to provide bounds to the execution time behaviour of software programs of interest, attached to acceptance thresholds similar in nature to the probabilistic failure rates required for certified systems (e.g. $10^{-9}$ per hour of operation in airborne applications [2]). More precisely, for a software program on a given platform and a specified acceptance threshold, probabilistic timing analysis provides a probability distribution that bounds the probability that the execution time of that program may exceed a given value. We call this bound a probabilistic worst-case execution time (WCET) or pWCET for short. The user of this technique is interested in the exceedance value that has a probability of occurrence no greater than the required acceptance threshold.

To some extent it may seem counter-intuitive, using ‘probabilistic’ bounds in systems requiring high integrity. However, the reality is that probabilistic modelling is a close match to the nature of the system. Let’s assume an aircraft with a computing system on which PTA is to be applied. The mechanical parts in the aircraft are designed with a probability of failure in mind. Effects such as radiation, repeated physical stress and extreme temperatures introduce a low, but not null probability of failure for the computing hardware itself. Hence, the system as a whole has a distinct probability of failure. In that sense, timing failures can be considered just another type of failure that the system may experience. PROARTIS engineers a platform such that timing failures can be assigned a probability of occurrence per unit of time, perfectly fitting with current practice for other type of failures.

As explained in D1.2, PROARTIS pursues three approaches to probabilistic timing analysis (PTA): measurement-based probabilistic timing analysis (MBPTA), static
probabilistic timing analysis (SPTA), and a hybrid approach. The requirements
needed by these approaches are met within the project thanks to the hardware
randomisation described in D1.2. The requirements for each analysis are detailed
in Sections 2.2, 2.3 and 2.4 of this deliverable.
SPTA, described in Section 2.2, provides a tight distribution for the pWCET of a
single linear-path program and the pWCET is obtained by convolving the proba-
bility distributions called Execution Time Profile (ETP) describing each processor
instruction. This analysis is described originally in a journal publication to appear
in ACM TECS 2012 [5].
MBPTA provides a pWCET estimation that is proved safe and the pWCET es-
timation is obtained from end-to-end runs of a program by applying rare events
theory to the set of observed execution times values. Therefore MBPTA provides
pWCET estimation for the paths traversed during the measurement runs by re-
lating this information to the observed execution times values. The validity of
MBPTA is empirically established by comparing its results against the results of
SPTA. Note that MBPTA does not use SPTA in its internal functioning: SPTA
is used for comparison purposes only. The MBPTA is described in a ECRTS 2012
publication [6].
The applicability of the EVT (extreme value theory) projections given by MBPTA
only covers the observed paths used to feed MBPTA. The problem of providing
relevant input data is open (and beyond the purpose of this project). We alterna-
tively propose a hybrid solution that is based on a combination between MBPTA
and RapiTime [24]. We call this method Hybrid Probabilistic Timing Analysis
(HyPTA). This solution provides a tight pWCET estimation and it is built us-
ing similar technology to RapiTime, which uses a combination of static structural
analysis and execution time measurements of sections of code. HyPTA takes small
segments of code which correspond closely to source code blocks and it builds a
probabilistic envelope that is the pWCET estimate. The analysis as well as its
requirements are described in Section 2.4.
Based on the characteristics of the different PTA methods and the behaviour of
the underlying hardware as described in D1.2, we build the time composability
aspects of PROARTIS as described in Section 3. Results supporting our findings
are provided in Section 4. Note that pWCET estimates require both the hardware
and PTA techniques to be in place. Thus, results provided in this deliverable
summarise the work done in deliverables D1.2 and D3.4 itself.
In conclusion this document corresponds to the deliverable D3.4 and it contains
the presentation of three probabilistic analyses: SPTA, MBPTA and HyPTA and
the time composability aspects emanating from them. The purpose of SPTA is to
prove the safeness of MBPTA and it can be applied only to single path programs.
The application of SPTA to more complex programs is not possible at this stage
because of the absence of a formula generalising (to multi-path programs) the
theoretical results on which SPTA is based. MBPTA is a first realistic solution
for probabilistic timing analyses, but the current knowledge of relevant data for
a program does not allow obtaining pWCET estimates for paths of the program
that are not observed during the measurement phase. HyPTA reduces the path
coverage requirements by combining the knowledge of the internal structure of the
program and the MBPTA techniques.
2

Probabilistic Time Analysis
Techniques

2.1 Main notations and definitions

We denote a random variable\(^1\) by \( \mathcal{X} \) and the probability function of \( \mathcal{X} \) is represented as:

\[
\mathcal{X} = \left( x_j \mid P(\mathcal{X} = x_j) \right),
\]

where \( x_j \in [x_{\text{min}}, x_{\text{max}}] \) and \( j \in [0, k] \) with \( k \in \mathbb{N}^* \) the number of values that the random variable \( \mathcal{X} \) has. The cumulative distribution function \( F_{\mathcal{X}}(x) \) is defined as

\[
P(x_{\text{min}} \leq X \leq x) = \sum_{y=x_{\text{min}}}^{x} P(\mathcal{X} = y).
\]

Independence and identical distribution: PROARTIS uses the notion of sequence of random variables which is independent and identically distributed. We provide here the definition of this notion.

Definition 1 (Independent Random Variables) Two random variables \( \mathcal{X} \) and \( \mathcal{Y} \) are independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other event.

Definition 2 (Identically distributed Random Variables) A sequence of random variables is identically distributed if all random variables have the same probability distribution.

We say that a sequence of random variables is independent and identically distributed (i.i.d.) if they have the same distribution function and they are independent from one another. A sequence of (fair) die rolls where each roll is a random variable is i.i.d., since the random variables describe the same event and the outcomes are obtained from independent events (as the outcome of one roll is independent of the outcome of another roll).

Convolution. PROARTIS documents also use the notion of convolution of two random variables. The convolution of two random variables, as defined below, is

\(^1\)We utilise calligraphic letters to denote random variables
the mathematical corresponding term for the convolution of two ETPs, as defined in D1.2. This operation between two random variables may be done only if the two variables are independent.

**Definition 3 (Convolution)** The convolution $\odot$ of two independent random variables $X$ and $Y$ has as result a random variable $Z = X \odot Y$ with the probability density $f_Z(z) = \sum_{x=-\infty}^{+\infty} f_X(x) f_Y(z - x)$. It is known that $P(z_1 \leq Z \leq z_2) = \sum_{z=z_1}^{z=z_2} f_Z(z)$.

**Probabilistic envelope.** The hybrid approach we propose at the end of this section is based on the notion of probabilistic envelope that we define below.

**Definition 4 (Probabilistic envelope)** Let $X_1, X_2, \ldots, X_n$ be $n$ random variables. $Y$ is the probabilistic envelope of the $n$ random variables if for any point $x_i \in [x_{i}^{\min}, x_{i}^{\max}]$, $P(Y = x_i) \geq P(X_i = x_i)$.

Definition 4 extends the notion of stochastic dominance [11] to $n$ random variables.

### 2.2 SPTA

In *Static Probabilistic Timing Analysis* (SPTA), execution time probability distributions for individual operations, or components, are determined ‘statically’ from a model of the processor or software. The design principles of PROARTIS ensure that it makes sense to derive execution time distributions for these operations, and to make the assumption that the probabilities for the execution time of each instruction are independent. More precisely SPTA needs to know the ETP for each instruction and the independence hypothesis to hold for the execution time of all instructions as described in D1.2. As mentioned in that deliverable, as the probability each memory operation taking a given latency depends on the execution history, SPTA is applied per path, recovering the i.i.d. property. Individual path must be combined to provide pWCET for a multi-path program.

SPTA is performed by calculating the convolution ($\odot$) of the discrete probability distributions which describe the execution time for each instruction on a CPU; this provides a probability distribution, or Execution Time Profile (ETP), representing the timing behaviour of the entire sequence of instructions (see Figure 2.1). As explained in D1.2, ETP is expressed by a pair (timing vector, probability vector), where the timing vector of an instruction enumerates all the possible execution times of that instruction and the probability vector provides the associated probabilities of each timing value.

Applying this technique for all instructions yields a new execution time distribution for the entire sequence of instructions, which includes all possible execution times for that sequence. By choosing a cut-off probability (e.g., $10^{-9}$) and measuring where the inverse cumulative distribution function (the exceedance function) falls below this level, we can read off a pWCET estimation with a given confidence that we will not exceed that value.

---

---

\[2\]We apply the term Static to all analyses that derive the a-priori probabilities from an abstraction of the system.
The convolution of two ETPs requires only one assumption to be made about the input data: that the probability that an instruction causes a cache miss must be independent of the sequence of hits and misses which have occurred due to previous instructions. Note that this does not require that the probability distribution for an instruction is independent of the sequence of preceding instructions, only that the actual outcome of an event (a cache hit or miss) will not affect the calculated probability distribution for following instructions.

Dependences between instructions may be modelled by creating profiles for short sequences of instructions which represent all the possible timing interactions between them. We aim to reduce the number of such dependences so that the complexity of combining instruction sequences is reduced. This dependence is identified in D1.2 and it is called dependence on execution history.

For this explanation we use a basic model of the processor with instruction and data cache only. We further consider fully-associative random-replacement caches implementing Evict-on-Access miss policy. However, the set-associative cache designs presented in D1.2 completely accomplish the principles to make probabilistic timing analyses work. In Section 4 and D4.3 we show results of the probabilistic techniques developed in this section when applied to programs running on an architecture including the designs presented in D1.2.

The major benefit we have from our time-randomised cache design over conventional ones is that pathological eviction patterns can be probabilistically bounded. It also reduces the amount of information needed for modeling the cache behaviour to only the number of memory accesses, not their locations.

In our time-randomised cache configuration based on a evict-on-access random replacement policy, the probability of evicting a given entry on any single access is $\frac{1}{N}$, where $N$ is the number of cache entries. From [26] we may calculate cache-survival probabilities based on the number of cache misses, $K$, between two consecutive accesses to the same cache entry using the formula:

$$P(\text{hit}) = \left(\frac{N-1}{N}\right)^k$$

(2.2)

In our case where the cache configuration is based on an evict-on-access replacement policy, every access causes an eviction; therefore, we define $K$ to be the number of memory accesses between two consecutive accesses to the same cache entry, including the access for which we are computing $K$. $K$ is referred to in this paper as the ‘reuse distance’ of an address. The value of $K$ must be computed separately for each memory access, as the reuse distance depends on the address.

Equation (2.2) does not satisfy our independence requirement: given two consec-
utive accesses to an address $A$, the hit probability is lower for the second access if all the intermediate accesses were hits than if they were misses. This is due to conditional probability: if we know that address $B$ was not evicted because we observe a hit, then the chance that we evicted $A$ instead is higher than if we observe a miss. Determining the exact hit probability has exponential cost in the size of $K$; however, a pessimistic lower bound on hit probability can be computed by applying Equation (2.3). We assume that there is an upper bound on the amount of information which could be known about the cache contents over the reuse distance. We take the case where all of the addresses accessed are unique and these entries are known to be in the cache for all of the reuse distance, reducing the effective cache size available to $A$ from $N$ to $N - (K - 1)$. Therefore, independence is ensured and distributions obtained from this formula may be convolved.

$$
P(hit) = \begin{cases} 
\left( \frac{N - (K - 1)}{N - (K - 1)} \right)^K & \text{if } K < N \\
0 & \text{if } K \geq N
\end{cases} 
$$

(2.3)

Under this cache configuration, the only information that the analysis requires for each memory access is the reuse distance and not the full sequence and addresses of the previous memory accesses as required when analysing conventional cache designs such as LRU.

We consider an example of a sequence of instructions described by Listing 2.1, where $i_h$, $i_m$ are the probability of hit and miss in the instruction cache respectively; $d_h$, $d_m$ are the probability of hit and miss in the data cache; and the hit and miss latency is 1 and 101 cycles respectively.

```
0: \{(2, 101), \ (i_h \times d_h) , \ i_h \times d_m + i_m \times d_h , \ i_m \times d_m) \},
1: \{(2, 101), \ (i_1) , \}
2: \{(2, 101), \ (i_h \times d_h^2) , \ i_h \times d_m^2 + i_m \times d_h^2 , \ i_m \times d_m^2) \},
```

Listing 2.1: Execution time profile for a sequence of three instructions. Instructions 1 and 3 access memory, whereas 2 does not.

Each line contains the execution time profile for one instruction: the first tuple are the latencies in the event each possible hit/miss combination occurs. The possible latencies of a non-memory operation are $(X+1, X+100)$ depending on whether the instruction cache access is a hit or not. In the case of the second instruction in Listing 2.1, $X$ – the inherent instruction latency – is equal to 1. Memory operations, i.e. instructions 1 and 3 in Listing 2.1, have three possible latencies: $(2, 101, 200)$. The numbers following this group are the probabilities of observing the corresponding latencies.

### 2.2.1 Reducing the complexity of SPTA

The calculation of SPTA is based mainly on convolution whose complexity is linear dependent of the number of possible values for each random variables. The complexity of SPTA may be reduced by decreasing for each random variable $X$ the number of values $k$ as long as the constraints are met. This process is called
Figure 2.2: The example distribution with two different basic re-samplings.

re-sampling. An example is illustrated in Figure 2.2. A distribution is described by the first line of the figure and two results of two different re-sampling techniques are illustrated by the last two lines of the figure.

**Definition 5 (Re-sampling)** Given $X$ a random variable with $n$ values, we call re-sampling to $k$ values or $k$-re-sampling the process of reducing the initial distribution $X$ from $n$ values to just $k$ values, while not introducing optimism with respect to the initial distribution.

We present two of the re-sampling techniques that we have implemented and we currently use in our tool developed at Rapita Systems [24]. These techniques have been presented in a paper sent to RTAS'2012 conference.

The first technique is a ‘uniform spacing’ re-sampling technique that is frequently used in practice and it provides good results. The second one is a more complex technique that provides better results, i.e., introduces far less pessimism, compared with other techniques that we have studied.

**Uniform Spacing Re-sampling**

We present here a first re-sampling method called uniform spacing. The re-sampling is done by choosing equally distanced values out of the initial distribution. Algorithm 1 describes the steps of this re-sampling technique. Here $C$ is the initial distribution, $C^{new}$ is the distribution obtaining after the re-sampling process. We denote by $C.size$ the number of values of $C$ and $C_j$ the $j$th value of $C$. The probability associated to the $j$th value of $C$ is denoted by $C_j.prob$.

**Reduced-Pessimism Re-sampling**

A uniform selection of values for re-sampling will not result in an optimum reallocation of probability mass. Some of the selected values will accumulate a disproportionate share of the overall probability. This can lead to large amounts of
Algorithm 1 Uniform Spacing Re-sampling Algorithm

INPUT $\mathcal{C}$ a distribution and $k$ the number of values to be selected

OUTPUT $\mathcal{C}_{new}$

$m = 1;$
$p = 0;$
$q = \text{ceil}(\mathcal{C}.\text{size}/k);$ 

for $i = 0; i \leq \mathcal{C}.\text{size}; i ++$ do
$p = p + \mathcal{C}_j.\text{prob};$
if $i \mod q = 0$ or $i = \mathcal{C}.\text{size}$ then
$\mathcal{C}_{new}.\text{value} = \mathcal{C}_j.\text{value};$
$\mathcal{C}_{new}.\text{prob} = p;$
$p = 0;$
$m = m + 1;$
end if
end for

pessimism in the re-sampled profile, and this pessimism is compounded by later applications of re-sampling which may occur during response time computation.

We propose an improved algorithm for the selection of re-sampling values which we call reduced-pessimism re-sampling. It works by considering ranges of values and calculating the pessimism which would be introduced if the range of values were to be aggregated into a single entry with the highest value in the range taking all the probability mass.

Algorithm 2 describes the method we use for computing the pessimism, which is based on the relative probability ‘weight’: the execution time multiplied by the probability.

Algorithm 3 provides the method by which re-sampling values are selected. Starting with the entire distribution, ranges are examined to identify the range with the highest pessimism. The selected range is then split into two sub-ranges.

Algorithm 2 Algorithm to compute the pessimism associated with replacing a range with a single value

INPUT $\mathcal{C}$ and $(x, y)$ a range of values

OUTPUT $p$ pessimism

$\text{shifted} = \sum_{i=x}^{y} \mathcal{C}_i.\text{probability} \times \mathcal{C}_i.\text{value};$

$\text{original} = \sum_{i=x}^{y} (\mathcal{C}_i.\text{probability} \times \mathcal{C}_i.\text{value});$

$p = \text{shifted} - \text{original};$

Examples of re-sampled distributions after applying the two re-sampling techniques are presented in Section 4.

We studied the efficiency of our re-sampling techniques by comparing calculations of response time analysis for set of tasks that have pWCET estimations $\mathcal{C}$ and pWCET described by re-sampled $\mathcal{C}_{new}$. In [7, 20] the authors provide the calculation of the probabilistic response time of a job under a preemptive uniprocessor fixed-priority scheduling policy as given by Equation (2.4).

$$\mathcal{R}_{i,j} = \mathcal{W}_{i,j}(\lambda_{i,j}) \otimes \mathcal{C}_i \otimes \mathcal{I}_i,$$  \hspace{1cm} (2.4)
Algorithm 3 Algorithm for selecting values which create the least pessimism when re-sampling

INPUT $\mathcal{C}$ a distribution and $k$ the number of values to be selected

OUTPUT $\mathcal{C}_{\text{new}}$

$Q = \emptyset$; // Priority queue
$r = (1, \mathcal{C}.\text{size})$; // Full range of $\mathcal{C}$
$p = \text{pessimism}(\mathcal{C}, r)$;
$Q.\text{add}(r, p)$; // Add $r$ to $Q$ with priority $p$

while $Q.\text{size} < n$ do
    $r = Q.\text{remove}_\text{first}$;
    $(a, b) = \text{split}(r)$; // Split into two equal sub-ranges
    $Q.\text{add}(a, \text{pessimism}(\mathcal{C}, a))$;
    $Q.\text{add}(b, \text{pessimism}(\mathcal{C}, b))$;
end while

$\mathcal{C}_{\text{new}} = \text{resample}(\mathcal{C}, Q.\text{upper bounds})$;

where all the variables are random variables and the release time $\lambda_{i,j}$ of the job $\tau_{i,j}$ is deterministic. Here $W_{i,j}(\lambda_{i,j})$ is the backlog at time $\lambda_{i,j}$ as the workload of higher priority jobs than $\tau_{i,j}$ that have not yet been processed immediately prior to $\lambda_{i,j}$. Equation (2.4) is solved iteratively in [20]. $\mathcal{C}_i$ is the execution time of job $\tau_{i,j}$ and $\mathcal{I}_{i,j}$ is the interference in $\tau_{i,j}$ of all higher priority jobs than $\tau_{i,j}$, $hp(i)$, released at or after $\tau_{i,j}$, $\mathcal{I}_i = \sum_{\tau_{k,l} \in hp(i)} C_k$.

We implemented the calculation of a solution for Equation 2.4 in a tool (Probabilistic Response Time Analysis) that is presented in the deliverable explaining the tool-chain, D3.5.

2.3 MBPTA

Let us assume a PTA-compliant processor and program for which we would like to provide pWCET estimates. Further assume that the execution of the program is observed $n$ times and the execution times are collected. If those execution times can be modelled with independent and identically distributed (i.i.d.) random variables [18], then the pWCET of a program can be derived by collecting enough observations to construct an Empirical Cumulative Distribution Function, ECDF. From the inverse ECDF it is possible to compute an estimated pWCET as $\overline{p\text{WCET}}$ for a desired probability. Such an approach has an important requirement that we must have a good model of the tail-end behaviour of the ECDF; this requirement may be fulfilled either by developing a suitable Monte Carlo simulation model or by making a vast number of observations to model these low-probability events. We are not aware of any Monte Carlo model able to provide sound pWCET and obtaining a vast number of observations is not practical (it is even difficult to prove the sufficiency of observations in the latter case). Alternatively, we use mathematical methods to predict, out of a small enough number of samples, pWCET bounds for exceedance probabilities smaller than $10^{-n}$, where $n$ is a required level of confidence. For MBPTA we have explored a mathematical method based on Extreme Value Theory (EVT). This theory estimates the probability of occurrence of extreme
values, whether high or low, which are known to be rare events [8]. More precisely, EVT predicts the distribution function for the maximal (or minimal) values of a set of observations, which are modelled with random variables. The EVT theory is analogous to Central Limit Theory [11] but instead of estimating the average, EVT estimates the extremes [14]. As in our work we are interested in the high values that bound the pWCET, we consider the EVT prediction for maximal values of a set of observations.

EVT has previously been applied to the WCET estimation problem in [8,17]. The main limitations of these two papers, which are shown in [15], are the following: the assumption of independent and identically distributed random variables, and fitting a continuous distribution to discrete values. Within this section we correct the application of EVT to WCET estimation by proposing solutions to these limitations. Moreover, we also provide an alternative to the $\chi^2$ test in [17] for fitting the Gumbel CDF (cumulative distribution function). Finally, we show how to apply EVT to multi-path programs.

The main result of EVT is provided in Theorem 1, where $F$ denotes the common distribution function of $n$ random variables.

**Theorem 1** [14] Let $\{X_1,X_2,\ldots,X_n\}$ be a sequence of i.i.d. random variables and let $M_n = \max\{X_1,X_2,\ldots,X_n\}$. If $F$ is a non degenerate distribution function and there exists a sequence of pairs of real numbers $(a_n,b_n)$ such that $a_n \geq 0$ and $\lim_{n \to \infty} P(M_n-b_n/a_n \leq x) = F(x)$, then $F$ belongs to either the Gumbel, the Frechet or the Weibull family.

In our approach, random variables are used to model the execution time of programs. In order to apply Theorem 1 two main hypotheses must be verified: that the $n$ random variables are independent and identically distributed and the existence of the sequence of real numbers $(a_n,b_n)$. The Gumbel, Frechet and Weibull are particular cases of the Generalised Extreme Value (GEV) distribution which has the following CDF:

$$F_\xi(x) = \begin{cases} 
  e^{-(1+\xi x/\sigma)}} & \xi \neq 0 \\
  e^{-e^{-x/\mu}} & \xi = 0
\end{cases}$$

GEV is defined by three parameters: shape ($\xi$), scale ($\sigma$) and location ($\mu$). By determining these three parameters, we prove the existence of the sequence of real numbers $(a_n,b_n)$. If the shape parameter $\xi = 0$ then $F_\xi$ is a Gumbel distribution, which is the distribution we use in this analysis.

When EVT is applied, we must determine its three parameters [13] and hence the appropriate CDF (Gumbel, Frechet or Weibull) is selected. To that end, a goodness-of-fit test is needed. However, not all tests are appropriate for fitting extreme values. For instance, in [17] the authors use the $\chi^2$ test which is known to perform incorrectly for extreme values in any classical test [13]. One test that is proven correct when fitting the Gumbel CDF is the exponential tail (ET) test [13]. The ET test assumes that the data set is modelled by $n$ i.i.d. random variables $X_1,\ldots,X_n$ described by the CDF $F$. The ET test compares this to the Gumbel CDF: $H_0 : F = F_\xi$. If the test rejects the hypothesis $H_0$, then the hypothesis $H_1 : F \neq F_\xi$ must be true.
The test is based on the comparison of two estimators for an extreme quantile of the law data. For our problem we restrict ourselves to models for which the law of excess follows an exponential law, that is to say laws in the domain of attraction of Gumbel (DA (Gumbel)). Therefore we choose a number positive $p_n \leq \frac{1}{n}$. We estimate the quantiles $1 - p_n$ using a parametric estimation. By using the parametric model, we test the suitability of the tail distribution, we obtain the following parametric estimator:

$$\hat{q}_{\text{param},n} = F_{\hat{\theta}_n}$$

(2.5)

where $\theta_n$ is a consistent estimator (e.g., the maximum likelihood estimator) if there are parameters of the distribution function $F$ of this model.

The confidence interval for the true quantile $q_{1-p_n}$ is then calculated as:

$$IC_{\text{re},n} = [\hat{q}_{\text{ET},n} + d_n \pm \hat{\sigma} \frac{ln\left(\frac{m_n}{n\hat{p}_n}\right)(z_1 - \alpha/2)}{\sqrt{(m_n)}}]$$

(2.6)

where $z_1 - \alpha/2$ is the quantile of order $z_1 - \alpha/2$, $\hat{\sigma}$ is the empirical average of excess (a natural estimate of $\sigma_n$; $d_n$ is an approximation (first order) of the through approximation; $m_n$ is the number of excesses.

We reject $H_0$ if $\hat{q}_{\text{param}} \notin IC_{\text{re},n}$ and accept $H_0$ otherwise. The acceptance of $H_0$ indicates that the data follows a Gumbel distribution.

The details for applying ET to our data are provided in Section 4.3.1.

### 2.3.1 Steps in the application of EVT

There are two main steps in the application of EVT: grouping and the data fitting. **Grouping:** The objective of this step is to collect, from the original distribution, the values which fit into the tail, and hence can be modelled with the Gumbel distribution. The values used as input for EVT are grouped into blocks of equal length $m$ by applying the block maxima method. The maximum value observed in each block constructs a new sample, the block maximum series.

The size of the blocks determines the portion of the original distribution that is considered the tail. The larger the block size, the better the fit to the tail; however, increasing the block size results in fewer blocks and thus fewer values in the final sample. In the extreme case, if a single block is used, we will have only one block maximum value corresponding to the maximum of the original distribution. Conversely, if we use as many blocks as elements in the original distribution, the distribution of the block maximum values will gather the entirety of the original distribution, rather than capturing the essence of the tail.

The theoretical basis of the block maxima method is built on the following theorem which says that if some data fit the Gumbel CDF, then the maxima of any blocks obtained from these data will fit the same Gumbel CDF.

**Theorem 2** [9] [Grouping] Let $X$ be a random variable. If the $n$ variables $X_1, X_2, \ldots, X_n$ obtained from $X$ fit a GEV, then the maxima of any blocks obtained from $X$ fit the same GEV.

**Distribution fitting:** If the ET test shows that our observations follow a Gumbel distribution, we next estimate the two remaining parameters, $\mu$ and $\sigma$ using the
block maxima series. Their practical estimation is obtained by applying linear regression to the QQ-plot of our data [10]. A QQ-plot (quantile plot) is a plot of the empirical quantile values of observed data against the quantiles of the standard form of a target distribution. Given that the block maxima values follow a Gumbel distribution, then the points on the QQ-plot form a straight line [10]. The slope and the intercept of the best-fit line through these points can be used as estimators for the Gumbel $\mu$ and $\sigma$ parameters, respectively.

### 2.3.2 EVT for single-path WCET

When applying EVT to the problem of providing $p_{WCET}$ estimates it is necessary to ensure that the assumptions made by EVT hold, so that the estimates obtained are sound. As we show in this section, these hypotheses have implications for the way experiments are run and the behaviour of the system. In particular, EVT assumes that the execution times of the system are independent between runs, and identically distributed.

We assume that the program for which a $p_{WCET}$ is to be estimated contains no system calls and that it is run in isolation. This is not a fully realistic scenario for production systems, but the assumptions match those used for current WCET analysis techniques; the interaction with other tasks and the operating system is taken into account during system integration (Section 3 and Deliverable D2.2). For this section, we further assume that the program contains a single-path of execution (in the next section we focus on multi-path programs).

In a fully deterministic system, a single execution path, when run from identical starting conditions, should have a single execution time. In practice this is not the case due to interference effects caused by hardware features such as DRAM refresh cycles, and in any case it is rare, or even impossible, to ensure that execution begins from identical starting conditions every time. Past approaches have attempted to use EVT to model this execution time variability, but there are no statistical guarantees about the nature of the variability. Rather than attempt to model such complex behaviour, our approach is to bring the behaviour closer to the model, thus ensuring that the variability in execution time conforms to distributions which can be modelled with statistical methods such as EVT.

By ensuring that timing behaviour of each source of variability is independent, the combination of latencies introduced will converge on average to a closed form distribution. In most cases this is the normal distribution due to the finite variance of the input distributions, according to Central Limit Theorem. Sources which cannot be modelled by independent random variables must be considered to always take their upper bound.

**Input Data Dependence.** For the single path case, with known, fixed input data on a simple architecture, we can exactly compute the distribution of execution times which will be observed when the path is executed by using a static probabilistic model of the hardware behaviour [5]. This provides an excellent way to evaluate the behaviour of EVT when applied to a time-randomised architecture, and gives greater confidence that the method is still sound when applied to more complex architectures.

In order to perform measurement-based analysis, the user must provide a range of input data to the program, designed to stress the program and produce worst-case
behaviour. The choice of the data may affect the timing behaviour of the software, even if it does not affect the path taken. For example, dividing by some numbers takes more cycles than others, and data structures which contain pointers may refer to the same or different memory locations depending on the input, which affects the memory access pattern, and hence cache latencies.

Users cannot be asked to provide the ‘worst’ or even statistically representative data during the testing phase; they do not know (in general) what this data will be at deployment time, so we must design the platform in a way which prevents any temporal benefit being taken from input data, this is done with time-bounded resources as presented in D1.2.

There are hardware proposals which force operations, such as divisions, to always take their upper-bound execution time, which is a safe (pessimistic) assumption [22]. In [22] the authors introduce a ‘Worst-Case Mode’, that simply delays each request to a resource until its maximum execution time is observed. This is configurable by software and once active this technique is transparent to the executing programs. A PTA-friendly processor architecture has to incorporate such a technique. It will be activated during the collection of execution times and can be switched off during deployment.

A similar solution can be applied to data-dependent memory accesses which must be flagged so that effects due to the input data can be accounted for. Most current Instruction Set Architectures (ISA) include some hint bits in the operation code of memory operations, which are used, between others, to advise the processor whether to cache some data or to activate the data prefetcher. In the former case, if the designer detects that the data for a given load is going to have a single use, it activates this bit for the opcode of the load. Analogously, in our case, input data dependent memory operations have to be identified in the object code and flag them by means of these hint bits to prevent cache or other performance-improving features being used for these instructions.

**Platform Requirements.** As described in D1.2, our EVT-based MBPTA approach provides a pWCET estimate attached to an exceedance probability. Our approach to make this hold is to ensure that all sources of execution time variation in the system must either be safely upper-bounded or have a true probabilistic behaviour such that their measurements on target capture the outcome of the events that make the execution time vary.

Our approach to this end is to randomise the timing behaviour of the hardware resources with latency too high to upper-bound [5] and to upper-bound those whose incurred pessimism is acceptable, as in the case of the input data that we explained before. Once that is done, the only events that can affect the execution time are random. Consequently, the observed frequencies of each execution time represent probabilities with a confidence level captured by the EVT method. Hence the system behaviour observed during testing can be used to tell its behaviour during actual operation. A more detailed description of the hardware requirements can be found in D1.2.

**Independent and Identically Distributed Observations**

In order for the i.i.d. properties to hold in our case, the architecture we propose must ensure that each execution run is independent. For the purposes of this experiment, we will ensure that runs are independent by forcing the state of the hardware to be reset before each run begins.
Identical distribution at the path level requires only that the latencies which make up the total execution time are independently chosen each time a path is executed. Each possible execution time that a path can take must have an associated probability. Note that this does not require that the probability distribution for an instruction is independent of the sequence of preceding instructions, only that the observed timing behaviour in one run does not affect the timing behaviour of other execution runs.

Continuous vs. Discrete Functions: minimum number of observations

In [15] it is shown that using the Gumbel distribution function, or any continuous function, to approximate a discrete function of the execution time values produced by a program, could yield unsafe results which do not bound the execution time from above at all points. In [15] there are proposed different solutions to this problem by either fitting the continuous function to the upper bound of the exceedance distribution function and thus overestimating the discrete function, or by adding an offset to the distribution which accounts for this effect. For instance for the single path we consider here, the maximum offset required is likely to be a small number of cycles.

We introduce an overestimation of the discrete function when applying EVT using the block maxima method before the distribution fitting is applied. This produces a shift of the distribution, which makes the worst-case estimate pessimistic when compared to the ‘real’ distribution of execution times in our architecture. In order to obtain high confidence in such a distribution, we require that the number of observations is sufficient to produce an accurate description of the program’s execution time. We call this the **minimum number of observations**, and we define it as follows:

**Definition 6** For a given sequential program $P$ and an architecture $A$, $n$ is the **minimum number of observations** if there does not exist $m \in \mathbb{N}$ where $m > n$ such that the Gumbel CDF obtained for $n$ observations is an upper bound for the WCET of $P$ and also exceeds the CDF obtained for $m$ observations.

We obtain the **minimum number of observations** by comparing the variation in the EVT tail projection as we consider an increasing number of runs. We follow an iterative process where 1) we start by running $N_{\text{current}} + N_{\text{delta}}$ times the program under study on the target platform. 2) We then, make two tail projections with EVT one using $N_{\text{current}}$ runs and the other using $N_{\text{current}} + N_{\text{delta}}$. 3) If the difference between the two EVT distributions is below a given difference threshold we stop the process. If it is over the threshold, we consider all previous runs as $N_{\text{current}}$, or in other words we make $N_{\text{current}} = N_{\text{current}} + N_{\text{delta}}$, and we make $N_{\text{delta}}$ more runs, consider the part of the sample and repeat the process from step 2.

As we keep adding more runs to $N_{\text{current}}$ the proportional effect that $N_{\text{delta}}$ new runs has on the EVT reduces, so a convergence of the algorithm is ensured. On the other hand, there is no guarantee that there is a strict convergence so we can have some local minima. In order to take into account this non strict convergence, we change the algorithm so that instead of stopping the process as soon as the difference between the two EVT distributions is below a given difference threshold, we stop the process when this is the case in several consecutive iterations. For the benchmarks used in this paper, considering 5 consecutive iterations below the threshold is sufficient to deal with the hysteresis introduced by local minima.
The metric we use to compare the two EVT distributions (functions) is called the continuous rank probability score defined as
\[ CRPS = \sum_{i=0}^{+\infty} [f_X(i) - f_Y(i)]^2 \] [4]. This test is defined for any two distribution functions \( f_X \) and \( f_Y \) as long as they operate on the same value domain. In this paper we set as difference the threshold \( 10^{-1} \). This threshold has to be interpreted as the significance level \( \alpha \) in hypothesis tests. The lower its value, the larger is the confidence in the result. It is up to the user of our technology to set a proper value to this difference threshold. However, note that the lower the threshold, the higher the required number of runs (observations).

**Step-by-step application of EVT**

EVT is applied to single-path programs by following these steps for a number of rounds until a stable result is obtained:

1) **Collect observations** – The first task is to gather a number observations of the execution time of the program. We suggest starting with around 100 values. In each round, an additional \( N_{\text{delta}} \) observations are made and included in the data used for the next steps.

2) **Grouping** – Block maxima sampling is used to convert the measured frequency distribution into a worst-case distribution suitable for application of EVT.

3) **Fitting** – The Gumbel distribution which best fits the observation is estimated. For this we use parameter estimation based on [13]

4) **Comparison** – If this is not the first round, the 1-CDF for the current round is compared to the result of the previous round using the CRPS metric.

5) **Convergence** – When the CRPS metric reports values below a given threshold (0.1 in the examples in this paper) for a number of consecutive rounds (5), we consider the distribution to have converged, and no more observations need to be collected.

If the distribution has not yet converged, the process starts another round at Step 1.

6) **Tail extension** – The final parameters for the Gumbel distribution are used to compute the execution time values for probabilities down to a lower bound. In our examples we use \( 10^{-16} \) as this is the limit of precision for floating point numbers in our prototype tool. Greater precision can easily be obtained using an arbitrary-precision floating point implementation such as MPFR [12].

2.3.3 Multi-path programs

The application of EVT MBPTA to multi-path programs is straightforward in principle, with three important requirements: 1) the compound distribution resulting from multiple paths must provably exhibit the required i.i.d. properties; 2) a sufficient number of observations must be made for each path; and 3) the result only applies to the set of observed paths. In the following we discuss each of them in isolation.

**Independent and identically distributed observations** The i.i.d. property that already holds for single-path programs may be preserved for multi-path programs by either selecting inputs randomly when running the measurement runs and grouping them sequentially, or testing with all inputs and selecting results randomly, without replacement, when performing grouping. We assume that there is a direct and traceable correspondence between the input (data and state) to a measurement run and the path taken by the execution.
**Minimum Number of Observations.** EVT is insensitive to the relative frequencies of different paths in the measured data: the process of extracting the block maxima distribution allows the data corresponding to worst-case paths to dominate the result. There is however a requirement that each path must have been observed a minimum number of times to allow the behaviour to be sufficiently well characterised. One possible way to ensure this property is to execute each input from the test set at least this minimum number of times.

**Path Coverage.** Extending this problem in another direction, the result is only valid for the paths which are observed. Claims on the pWCET for untested paths cannot be made, as the execution times for those paths would no longer be drawn from an identical distribution. This inherently couples the concept of i.i.d. with the path coverage achieved by the input data.

The issue of path coverage introduces a problem for the usefulness of the result, as full path coverage in complex applications is infeasible: the combination of loops and branches creates an explosion in the number of possible paths through the program. Therefore we must impose a caveat on the result obtained by EVT: that it is valid only for a set of ‘observed paths’, that is, those paths covered by the tests.

In order to apply our method in a realistic scenario therefore, it will be necessary to strike a balance between precision and practicality. This may be achieved by limiting the scope of the program being considered for pWCET estimation. For instance, the units over which EVT may be applied could be defined as the largest partitions of the control-flow graph over which full path coverage has been observed. Once estimates have been calculated for these units, the results may be combined using a tree-based composition, as in [3]. This allows the testing burden to be reduced, at the expense of increasing pessimism in the overall estimate. To further improve applicability: (1) we recommend that the measurement runs are performed as part of functional testing, so that our method attains no less path coverage than that required for functional testing, at the added cost of achieving a minimum number of observations per path, which may be larger than that required by functional testing alone; (2) we note that the attained path coverage can be recorded with modest enhancement to the test environment and existing methods can be used to determine unobserved paths of interest, i.e., those that could cause longer execution time than that captured by the observations (e.g. [24]).

### 2.4 HyPTA. Rapita Demonstrator

At this stage of the project, the SPTA proposes a tight estimation of the WCET of a program at the cost of knowing the reuse distance of all instruction and data accesses needed to derive the ETP for each instruction. Our SPTA implementation only works with single path programs. Fortunately the MBPTA is able to provide WCET estimate for multi-path programs and this regardless of the knowledge on the architecture or the program. The limitation of the MBPTA is related to the path coverage. MBPTA is able to estimate WCET for parts of the program that were observed during the measurements. This problem is solved once we know a relevant input data for a program. Unfortunately obtaining such set is an open
problem and it is beyond the purpose of this project. We therefore propose a hybrid probabilistic timing analysis based on the combination between the MBPTA and the knowledge of the internal structure of the program. We call this approach HyPTA.

The basic idea of HyPTA is to approach the pWCET estimation problem in the same way that RapiTime currently uses a combination of static structural analysis and runtime measurements of execution time and loop bounds to generate a WCET estimate. Therefore HyPTA is a “bottom-up” approach while MBPTA is a “top-down” one.

The HyPTA alternative takes small segments of code which correspond closely to source code blocks. The distributions for these segments contain only one path, and so are not directly affected by the path the program takes, although there are indirect effects from the hardware execution timing variability (i.e. cache state).

We overcome the problem of hardware variability in the same way as in the MBPTA approach. By applying the EVT tail extension method to the individual code segments, it is possible to obtain an upper bound on the behaviour of that block.

Once sufficient samples have been collected, the tail-extended profiles are then combined using the structure of the code to obtain an overall WCET estimate for the entire program, including synthetic paths which may not have been tested.

This is directly compatible with the existing RapiTime report structure, enabling a close integration into the Rapita Verification Suite (RVS) report viewer.

**Requirements.** Sequential statements or subtrees may be combined by simple convolution. In HyPTA we make an assumption that the distributions of two blocks may be combined using a simple convolution. For most straight-line sequences this may be true, since the aim of PROARTIS is to remove all hardware effects which would introduce dependences in timing behaviour on earlier execution of code (see D1.2). However, there is still some dependence on the contents of the cache, which may result in different timing behaviour when sequences are combined in new (untested) ways (see Section 2.2). In practice this effect is mitigated by the EVT tail extension, since applying this technique will extract the worst-case behaviour for that block before the convolution is performed, thus reducing the dependence between blocks, at the expense of increased pessimism. One good solution to this problem is to take measurements across larger units of code, encompassing several execution paths, and thus accounting for any dependences in the measurement.

The requirement for using such a sub-graph is that all paths in the graph have been adequately tested (several hundred runs of each path). While this is not a feasible requirement for an entire application, it is quite possible in isolated portions of code, such as library routines and other leaf-functions in the call graph.

Synthetic loop iteration does not affect the timing correctness of subsequent statements. In a deterministic cache design, once the cache has “warmed up” to a loop, the number of iterations does not have a great effect on the final cache state when the loop finishes. This means that a small number of loop iterations can be used to accurately predict the timing for a much larger number, and the timing behaviour of subsequent blocks is not affected.

In the PROARTIS cache design, the probability that something remains in cache depends on the number of instructions (and hence evictions) which have been performed since that item was last accessed. This means that increasing the iteration count synthetically might not account for the degraded cache state which would
be present after executing that number of iterations. The implication here is that the timing behaviour for blocks following a loop which has only been seen to execution 10 times on one path will be optimistic in the case where the loop actually iterates 100 times, as observed in another path. The problem is mitigated as the user makes additional tests to bring the observed software path execution closer to the predicted worst-case behaviour, which will reduce the size of any synthetically introduced error, and also improve the available data with which to make the analysis. A feedback step is crucial to inform the user about the quality of their testing, and to improve the estimate.

**Tree composition.** The tree composition is performed using a WCET expression, which is a script representing the structure of the program being analysed. Subtrees in the expression may be combined sequentially, in parallel (a conditional statement), or repeated (a loop). Sequential statements are combined using a simple convolution operation. Choices between two or more branches are resolved using a probabilistic envelope on the cumulative distributions of each branch. We believe this to be a safe upper bound, as it will always exceed any of the branches, regardless of the relative frequency with which they are taken.

Loops are computed by performing a convolution powering operation, which convolves the profile for one loop iteration with itself the maximum number of times the loop may iterate. As with deterministic hardware platforms, it is likely that the first iteration of a loop body will have different cache (or path) behaviour than subsequent iterations, so it may be desirable to “unroll” the first loop iteration. This is purely an analysis technique, and does not alter the code generated by the compiler.

**EVT Tail extension.** Each frequency distribution needs to be converted to a tail-extended probability distribution before the overall profile is constructed. This is accomplished by applying the EVT method used in MBPTA. Since within RapiTime we do not currently store the full sequence of execution times, but only the frequency distribution, a randomised sample is generated with the correct number of values in the appropriate frequencies before being passed to the EVT script. The EVT script returns the best-fit parameters for a Gumbel extreme value distribution, which must be converted into a discrete probability distribution with integer execution time values.

**Safeness and pessimism.** HyPTA is safe, but pessimistic. The safeness is guaranteed by the definition of the probabilistic envelope since the envelope contains all maximal values of the initial distributions of the basic blocks. In order to estimate how pessimistic the hybrid approach is, we compare in the results section the MBPTA result for a program on which we apply the HyPTA.

**Respecting the EVT hypotheses.** Since each block is a single path program then by applying the steps of the EVT as described in the part first of this section, we ensure that the hypotheses are met.

The HyPTA approach is being implemented into RVS/RapiTime. Early results using this technique are reported later. The implementation will be released in due course.
3

Time Composability aspects of PROARTIS

One of the major benefits brought forward by the software tiers running on top of PROARTIS hardware is that the timing behaviour of individual processor instructions is independent of execution history: this is a precious asset in the pursuit of time composability at application level. Independence in the timing behaviour of individual processor instructions already warrants time composability for instruction sequences, which is a great facilitator to correct and cheaper timing analysis. While this is an excellent result in itself, it does not immediately scale to considering software programs as units of composition, instead of processor instructions.

The industrial domains of interest to PROARTIS have three main requirements on time composability:
- the timing behaviour of individual partitions as analysed in isolation must not to be adversely affected when partitions are composed to form the system;
- the timing behaviour of individual threads as analysed in isolation not to be adversely affected when threads are composed to form a partition;
- the timing behaviour of individual threads as well as of individual partitions as analysed in isolation not be to adversely affected when they are composed with the Operating System in the full system configuration.

To meet those requirements, additional means must be devised to ensure that the concurrent execution of distinct software units cannot cause the timing behaviour of individual units to incur dependence on their interleaving (hence on the history of execution) or, if some dependence occurs, that it can be bounded from above with sufficient tightness, without needing to know much about the interfering software. The goal here is to achieve time independence at the granularity of software programs.

3.1 Time Composability in PROARTIS

Consider a unit of composition (UoC) B to which PTA is to be applied. A UoC is to be understood as a program unit that the user wants to be able to compose with others to form (parts of) the system. Following the PROARTIS PTA method,
this UoC is also the natural candidate of timing analysis. Further assume that this UoC makes no calls to the Operating System – or equivalently that the effect of such calls can be ignored as per the discussion that follows in D2.2 – and that the UoC includes – as if by inlining – all the finer-grained procedures that it calls. We want to guarantee that the timing analysis performed for B stays valid in the face of its composition with any other UoC, at the same level of abstraction\(^1\) during system integration.

### 3.1.1 Assumptions

We assume run-to-completion semantics for all UoC so that the interference effects resulting from pre-emption do not need to be covered in the analysis. In later subsections we elaborate more on this point.

We focus on the hardware aspects of the disturbance to time composability, in particular on the processor state effect on time composability. In the processor core architecture we have used all resources have constant execution time (no jitter) or are considered low-jitter as defined in D1.2. Only the different cache levels retain internal state and are considered high-jitter resources.

For this discussion we ignore the software state, which can be regarded as all the variables in the program that can influence the execution of the program under analysis. This influence materialises in determining the control flow path taken by the execution. We can ignore this issue because PTA, in either of its forms, applies to the paths that can be analysed and therefore – at this level of discussion – has no interest in selectively considering or excluding paths based on state analysis.

In a STA-friendly architecture, an easy yet pessimistic way to attain time composability for a UoC B is to compute its WCET assuming that the processor state is flushed prior to its every execution. This assumption is obviously pessimistic, but it has the advantage that it makes no assumption on the code executed before B.

At a first approximation, in a PTA-friendly architecture we can apply the same principle of flushing the processor state prior to the analysis of the UoC. This solution prevents the execution time of the UoC from being affected from previous history of execution, which makes it time composable (at the cost of the same pessimism as seen for the STA case). We now refine the notion of time composability from this brutal all-or-nothing notion.

### 3.1.2 SPTA

Let us consider two consecutive executions (instances) of a given UoC B in a statically scheduled timeline. Let us call them \(B_p\) and \(B_q\), where the subscript represents the ordinal execution number of the UoC instance, with \(q > p\), so that \(B_p\) precedes \(B_q\). The amount of local state that \(B_q\) can reuse from \(B_p\) and how much benefit this can cause on the execution time of \(B_q\) depends on: (1) how much internal reuse \(B_q\) makes, which we call intrinsic reuse and depends on the size of B’s working set; (2) the execution ‘duration’ of \(B_p\). The longer \(B_q\)’s the lower the relative effect of different initial conditions; (3) the size of the cache(s) that keep

\(^1\) The reader should appreciate that in this argument here we want to reason by layers and not across layers. Composition with the Operating System, which is across layers, is separately discussed in D2.2.
B\textsubscript{p}'s state in hardware; and (4) how many data (state) of B\textsubscript{q}'s working set are not evicted by the functions/code executed in between. We call this foreign code executed in between disturbing code and its effect state disturbance.

For STA we assume the Evict-on-Access (EoA) eviction policy for our random-placement replacement-caches. Under EoA, we know that the effect that any disturbing code can cause on B\textsubscript{q} is given by the number of accesses that the disturbing code performs in the instruction caches, which is given by the number of instructions; and the number of data accesses performed in the data caches. Each access to any cache in the architecture will age the data/instructions left by B\textsubscript{p} in cache, potentially affecting the execution of B\textsubscript{q}. When EoA is in place, we know that every access to the cache causes a random eviction, so the more the accesses of the disturbing code the more evictions hence the lower the probability that any data of B\textsubscript{p} survives in cache. Note that the terms age and survivability are opposed: the more the accesses of the disturbing function the higher the ageing of cache contents and the lower the survivability of data in cache. Reuse across UoC boundaries involves pairs of accesses, one being the access from B\textsubscript{p} and the other from B\textsubscript{q} in each pair. Any such pair consists of the last access to a particular datum \( r \) in B\textsubscript{p} and the first access to such datum in B\textsubscript{q}. The ageing of such datum (and thus the hit probability of the first access to \( r \) in B\textsubscript{q}) depends on the reuse distance between those two accesses to \( r \) when B\textsubscript{p} and B\textsubscript{q} are executed consecutively without any disturbing code intervening between them. The reuse distance \( K \) corresponds exactly to the number of accesses occurring in between those two consecutive accesses to \( r \). If disturbing code were executed in between B\textsubscript{p} and B\textsubscript{q} and made \( d \) cache accesses, the reuse distance would increase by \( d \). Thus, given a cache of size \( N \), equation 3.1 provides a tight bound of the hit probability of an access based on its reuse distance when no disturbing code is executed. If disturbing code was executed in between, then the hit probability can be tightly upperbounded with equation 3.2, which corresponds to the aged version of that hit probability when \( d \) cache accesses are made by the disturbing code.

\[
P(\text{hit}_k) = \begin{cases} 
\left( \frac{N-(K-1)-1}{N-(K-1)} \right)^K & \text{if } K < N \\
0 & \text{if } K \geq N
\end{cases} \tag{3.1}
\]

\[
P(\text{hit}_k) = \begin{cases} 
\left( \frac{N-(K+d-1)-1}{N-(K+d-1)} \right)^{K+d} & \text{if } K + d < N \\
0 & \text{if } K + d \geq N
\end{cases} \tag{3.2}
\]

Hence, in order to probabilistically model the effect of such disturbing code on the pWCET estimation for B\textsubscript{q}, we abstract all the code that executes between B\textsubscript{p} and B\textsubscript{q} with the number of data and instruction accesses that this disturbing code performs, \( N_d \) and \( N_i \) respectively. This information is sufficient to compute an upperbounded and aged pWCET estimations for B\textsubscript{q}. In the extreme case, if B\textsubscript{p} and B\textsubscript{q} were executed sequentially, with no disturbing code in between, there will be no ageing effect, so that B\textsubscript{q} can achieve maximum reuse from B\textsubscript{p}'s state. We call this the non-aged pWCET estimation or best pWCET estimation for B\textsubscript{q}.

This probabilistic approach to time composability has very useful properties: If we compute the pWCET estimation for B\textsubscript{q} assuming that in between its execution and the preceding execution of B\textsubscript{p}, \( \max_d \) and \( \max_i \) data and instruction accesses occur,
then \( B_q \) is time composable with any software program for which its number of data and instruction accesses \( N_d \) and \( N_i \) are such that: \( N_d \leq \text{max}_d \) and \( N_i \leq \text{max}_i \). Note that, this notion of timing composability is oblivious to the particular location in memory where the data and instructions of the disturbing code are because our random-placement random-replacement caches break the structural relation between the address and location of cache contents. This is a major advance with respect to the state of the art: In the case of static analysis on top of conventional modulo placement LRU replacement caches, if such an approach is to be used, it would be necessary to track the addresses of each access to determine to which particular set it is performed, ensuring that the sequence of accesses per set is exactly the same in all disturbing code to be composed with \( B_p \) and \( B_q \). Eventually, if any code is to be bound with such a scheme, after as many accesses as the number of ways of the cache (typically between 2 and 64) no cache line can be guaranteed to be in cache and no probabilistic guarantee can be given. Instead, random-placement random-replacement caches allow us (1) to not make pessimistic assumptions about the cache set accessed; and (2) to reason probabilistically on the evicted cache contents.

Let us illustrate this with an example. Figure 3.1 shows the code and data of two disturbing functions, \( df1 \) and \( df2 \) that cause different state interferences on \( B_q \). Each entry represents a piece of data or an instruction of the disturbing function. The color represents the cache set on which the data instruction is mapped. For this example we assume a 4-set 2-way set-associative cache implementing modulo placement and LRU replacement. Under this cache configuration \( df1 \) has the following sequences of instruction accesses to sets 1, 2, 3 and 4 respectively: \( \{i1, i5\}, \{i2\}, \{i3\}, \{i4\} \). The sequences of data accesses are \( \{d1, d5\}, \{d2\}, \{d3\}, \{d4\} \). For \( df2 \) the sequences of instructions and data accesses to each set are: \( \{i4\}, \{i1, i5\}, \{i2\}, \{i3\} \) and \( \{d3\}, \{d4\}, \{d1, d5\}, \{d2\} \).

In the case of STA, given that the sequences of accesses per set are different for \( df1 \) and \( df2 \), they cannot be composed identically with \( B_q \). Hence, if we provide a WCET analysis for apply PTA to \( B_q \) assuming that \( df1 \) is executed between \( B_p \) and \( B_q \), the analysis for \( B_q \) has to be repeated if \( df1 \) is changed by \( df2 \).

In this same scenario if we use a random-placement random-replacement cache, given that \( df1 \) and \( df2 \) have the same number of instructions and data, the pWCET estimation we obtain for \( B_q \) assuming that \( df1 \) is executed between \( B_p \) and \( B_q \) is a valid upperbound even if \( df1 \) is replaced by \( df2 \). This happens because random-placement random-replacement caches break the dependence between the addresses and the location of data in cache.

Time composability can be easily achieved and each UoC can be analysed for timing in isolation as shown next: For the first occurrence in the static analysis we assume empty cache, that serves as worst initial state. For any subsequent execution in the time line we can compute the pWCET estimation assuming the cache state left by the previous occurrence. Right before running the analysis in the modelled cache stage we age each cache line in cache by \( N_d \) for data caches and by \( N_i \) for instruction caches, which can be done simply increasing by \( N_d \) and \( N_i \) the reuse distance of each cache line in the data and instruction caches respectively. We call the resulting pWCET estimation the \( N_d-N_i \)-aged pWCET estimation.
Figure 3.1: code and data for two disturbing functions. Each color represent the cache set in which each instruction/piece of data is mapped)

3.1.3 MBPTA

With MBPTA-EVT the pWCET estimations are obtained by analysis of the execution time of the program from several runs. Our objective is the same as with SPTA: obtaining pWCET estimations for a UoC \( B \) such that the second and following instances of that UoC can benefit from previous instances. This pWCET estimations have to be obtained in isolation making the minimum possible assumptions on the disturbing code that can be executed in-between those instances of the UoC.

With MBPTA our preferred eviction policy is Evict on Miss (EoM) rather than EoA given that EoM performs better and it is as simple to analyse as EoA. With EoM, every time there is a miss in cache a line is randomly evicted from the set in which the eviction is to take place. Given the functionality of the EoM policy the metrics that best captures the interference caused by disturbing code is not just \( N_d \) and \( N_i \). We are after a set of metrics \( me \) that characterise a program unit such that given two disturbing codes \( df_1 \) and \( df_2 \), we can prove that the cache ageing caused by \( df_1 \) is higher than that caused by \( df_2 \) if \( df_1 \) is worse (higher) than \( df_2 \) for all metrics in \( me \). That is, for \( 1 \leq i \leq N \) where \( N \) is the number of metrics to consider, if \( me^1_i \geq me^2_i \) the effect of \( df_1 \) is worse than the effect of \( df_2 \). Under this situation, if we obtain a pWCET bound for \( B_q \) under the composition scenario \( B_p df_1 B_q \) we know that such pWCET bound is a true upperbound value for the composition scenario \( B_p df_2 B_q \).

Our approach focuses again on the reuse distances of the instruction and data accesses of the disturbing code: \( me = \{rd_i, rd_d\} \). Given two disturbing codes, \( df_1 \) and \( df_2 \), we say that the former produces worse interference than the latter if each instruction and data access of \( df_2 \) can be paired up with an instruction and data access respectively of \( df_1 \) with higher reuse distance. The rationale behind this approach is that under the EoM policy, the higher the reuse distance of an access is the higher its miss probability and therefore, the higher the probability that it performs an eviction, which reduces the survivability of cache contents. In other words, if \( df_1 \) performs at least as many instruction and data accesses as \( df_2 \) and the miss probabilities for \( df_1 \) accesses are higher than those for \( df_2 \), \( df_1 \) upperbounds \( df_2 \) cache aging. An easy mechanism to check how the reuse distances for data and instructions for \( df_1 \) upperbound their counterparts for \( df_2 \) is shown in Algorithm 4. As a consequence, the pWCET estimation that can be obtained for \( B_q \) under the composition scenario \( B_p df_1 B_q \) bounds the pWCET estimation that can be obtained for the scenario \( B_p df_2 B_q \).
Algorithm 4 Algorithm to check whether a disturbing program df1 bounds another disturbing program df2

Check if df1 bounds df2

\(C\) = set of caches in the processor

for all \(c_i \in C\) do

\(r_1\) = reuse distances for df1 in \(c_i\)

\(r_2\) = reuse distances for df2 in \(c_i\)

if \(|r_2| > |r_1|\) then

return false

end if

\(r_1\)sort = sort \(r_1\) from higher to lower

\(r_2\)sort = sort \(r_2\) from higher to lower

for all \(r_2\)sort\(_j\) \(\in r_2\)sort do

if \(r_2\)sort\(_j\) > \(r_1\)sort\(_j\) then

return false

end if

end for

end for

return true

Below we show some examples of reuse distance vectors for df1 and df2. In each case we show when df1 upperbounds df2. For this example, we only consider data accesses, but the same considerations apply to instruction accesses.

- df1 bounds df2: \(rd_1 = \{7,5,3,2\}\), \(rd_2 = \{6,5,2\}\). \(7 > 6, 5 \geq 5, 3 > 2, 2 > \emptyset\). Hence, d1 upperbounds d2.

- df1 does not upperbound df2: \(rd_1 = \{9,8,7\}\), \(rd_2 = \{1,1,1,1\}\). \(9 > 1, 8 > 1, 7 > 1, \emptyset < 1\). No reuse distance of df1 is paired up with the last one of df2, so df1 does not upperbound df2.

Although the approach described so far is conceptually applicable and capable of producing upperbound values, modelling the reuse distances of all the disturbing code that may run between \(B_p\) and \(B_q\) is complex, path sensitive and laborious. Hence, instead of using the reuse distances, which would give the tightest pWCET estimations, we follow a different approach, which we call ACConly approach. ACConly uses the number of unique accesses (\(u_d\) and \(u_i\)) and requires the disturbing code df1 to perform such number of unique accesses. Unique access have reuse distances \(\infty\) and therefore, any disturbing code df2 with \(N_d\) and \(N_i\) data and instruction accesses such that \(N_i \leq u_i\) and \(N_d \leq u_d\) is safely upper-bounded by df1. For instance, if we wanted to bound any disturbing code (df2) with up to 6 memory accesses, a (df1) code with the following accesses would bound it: \(@A@B@C@D@E@F\).

Hardware/Software support Disturbing code can be created by hardware or software.

In the case of hardware, ACConly would only need a hardware mechanism, which we call \(n_{\text{random-evictions}}\) able to perform a given number of random invalidations
in the desired cache memory. For the first instance of the UoC we assume empty cache, that serves as worst initial state. For the rest of the instances we can compute its pWCET estimations under this scenario: we execute $B_p$ and $B_q$ consecutively. Right before running $B_q$ we use the processor support to perform $\text{max}_i$ and $\text{max}_d$ random evictions. The execution times of $B_q$ resulting from these runs can be used to provide a safe pWCET estimation for $B_q$ that is composable in the presence of any disturbing code with $N_i$ and $N_d$ instruction and data accesses such that $N_i \leq \text{max}_i$ and $N_d \leq \text{max}_d$.

In the case of the software solution, $\text{ACConly}$ can be implemented with small programs we call microbenchmarks. The way a microbenchmark must look like for $\text{ACConly}$ is such that it produces a sequence of accesses to each cache memory such that all of them access different cache lines. Since combining such an effect for data and instruction caches simultaneously is difficult, a first part of the microbenchmark can produce the evictions for data and a second part can do it for instructions. In particular, data evictions simply require a loop with a load instruction whose stride is equal or higher than the largest cache line in any data cache. For instance, if we use a data cache with 32-bytes cache lines and a data TLB (translation lookaside buffer) with 1KB pages, performing $N_d$ loads with a distance of at least 1KB ensures that no reuse occurs in any of the data caches. A similar effect can be produced for instructions by executing branches with a stride equal or higher than the largest cache line in any instruction cache.

**Tradeoffs** As mentioned before the potential benefit that $B_q$ can make of the data and instruction state left by $B_p$ is inversely proportional to the ‘length’ of the disturbing code. Here, length has to be understood as the number of accesses in the case of $\text{ACConly}$. To illustrate this point, let us assume a cache in which after the execution of $B_p$ all cache lines have a reuse distance of 1. Figure 3.2 shows for two different cache sizes the survivability of each line as we increase the number of unique addresses (and therefore also accesses) to the cache in the disturbing function. The cache sizes resemble L1 and L2 caches used in the results section. After 600 accesses to unique addresses in a 256-entry cache survivability (i.e. the

![Figure 3.2: Survivability as a function of the number of unique accesses in the disturbing code for a 1024-set and 8192 set cache](image-url)
probability that \( B_q \) founds a piece of data or instruction left by \( B_p \) is less than 10%. For a 4096-entry cache survivability also becomes negligible when the number of accesses to unique addresses is higher than twice the number of cache entries. Thus, if we can expect programs with around 10,000 accesses to be executed between different instances of \( B \), it is useless trying to exploit cache contents from previous executions if using the \( ACC\text{only} \) approach.

**Conclusion**  We have shown how different UoC, all of which reside at the application level, can be time composed for the purposes of PTA. For each eviction policy that we consider, EoA and EoM, we have shown that the amount of information required to characterise disturbing code and make the UoC time composable is relatively moderate: the number of data and instruction accesses for EoA and the reuse distances for EoM. We have also shown that the amount of information can be further reduced to only the number of accesses and the number of unique addresses (upper-bounded by the working set). This is in contrast with approaches based on deterministic architectures that require knowledge of all addresses for any potential disturbing code to determine which cache contents may be are reused across executions.
4

Results for standard benchmarks and Rapita Demonstrator

In this Section we show all the evaluation results we have obtained during this second phase of the project other than the results with the industrial cases studies, which are presented in D4.3. The results presented in this section are listed below:

- We have used three different processor setups to explain in detail the benefits of the PTA approach. For each of the configurations we start by doing an auto-evaluation of the MBPTA-EVT technique in which we determine the different parameters affecting the MBPTA-EVT behaviour, e.g. minimum number of runs (mnr) and block size. We also evaluate the increment suffered in the pWCET estimations as we decrease the target exceedance probability. When appropriate we compare MBPTA-EVT and SPTA techniques in terms of tightness of the pWCET estimations.

- We compare the average performance obtained with our random placement and replacement cache against a cache implementing modulo placement and random replacement.

- We provide some initial evidence of the behaviour of our MBPTA-EVT approach with respect to static timing analysis in terms of WCET bounds.

- We evaluate the demonstrator done by Rapita Systems, described in Section 2.4.

- We also provide evidence about the time composability properties of the PROAR-TIS approach. This results are complemented with the results obtained for the single-core case study, D4.3.

All the experiments presented in Sections 4.1 through to 4.9 involve – by design – no Operating System support. D2.2 provides experimental evidence in support of the achievement of the constant-time zero-overhead Operating System design described in that same section; the processor setup used for the experiments related with D2.2 is discussed directly in that section. Future work will look into how the current experimental results will be confirmed after full integration between the application and the Operating System.
4.1 Hardware Experimental Setup

We modelled a pipelined processor featuring different caches. We use three distinct variants of a PTA-friendly architecture. We refer to those variants as *icache-only*, *single-level* and *multi-level* respectively. These variants differ mainly in the cache hierarchy configuration, while the architecture of the processor core is the same in all 3 variants.

4.1.1 Common processor core architecture

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

- **Fetch**: Instructions are fetched in-order from the Instruction cache (I$) into the processor. If the architecture has Instruction Translation Lookaside Buffer (ITLB), it is accessed in parallel to translate pages;

- **Decode**: Instructions are decoded in one cycle. Decoded instructions are issued to the execute stage;

- **Execute**: Instructions are executed in one cycle, except for memory operations. Read operations access the data cache (D$) during this cycle blocking the execution of further instructions until data are fetched. The Data TLB is accessed in parallel with the D$;

- **Write-back**: The instructions commit in one cycle. Write operations update a write buffer so they do not block this stage (as long as the buffer is not full).

Instructions are issued in the program order. For in-order processors, no instruction can be issued until all previous instructions have been issued. This is frequently the case for CRTES as this feature facilitates timing analysis. For instance, the LEON4 processor [1], used by the European Space Agency, issues one instruction per cycle, in-order. Similarly, instructions finalise their execution in-order so that no instruction updates the state of the register file or memory until all older instructions have been committed. This commit policy is common in CRTES too. Whereas the cited LEON4 allows instructions to finalise out-of-order, most of them have a one-cycle latency, hence they typically commit in-order. We consider in-order finalisation because it simplifies our experimental implementation and also speeds up simulation. However, any other architecture compliant with the requirements defined in D1.2 could be deployed.

With in-order issue and finalisation we prevent instructions from using resources out-of-order in the final stage of the pipeline. This avoids timing anomalies by construction [25]. More complex processor pipelines can also be shown to be PTA-friendly. Execution units with out-of-order finalisation may be used with arrangements that can be proven to be free of timing anomalies, either making those resources contention-free after the execute stage (adding enough write ports to the register file) or offloading the responsibility for avoidance to the software side.

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

\subsection{4.1.2 Cache architecture}

Both instruction and data cache size is 4-KB with 16-byte line size. Associativities considered are 1-way (direct-mapped), 8-way (set-associative), 32-way (set-associative) and 256-way (fully-associative). Both caches, instruction and data, implement the random placement and replacement policies presented in D1.2. The random replacement logic implements the multiply-with-carry PRNG (pseudo-random number generator) [21]. The D$ is write-through, so that all store instructions are propagated to memory. The D$ is not-write-allocate, so that memory locations are not fetched to cache on a store miss.

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

The L2 cache size is 512KB with 16-byte lines. It is a 8-way 64K set cache implementing our random placement and random replacement policies.

Next, we describe the main differences among our three cache setups as well as the need to have each of them.

- icache-only: This architecture features only instruction cache. There is no ITLB, DTLB or data cache. The data accesses are assumed to always take one cycle.

The SPTA technique described in D2.2 provides a very tight approximation to the actual execution time distribution of the program, though this is achieved at the cost of being time consuming. This processor setup is simple enough so that we can compute in a reasonable time pWCET estimations with SPTA and show evidence that MBPTA-EVT pWCET estimations are tight to those provided by SPTA.

- Single-level: This setup features all first level cache memories: I$, D$, ITLB and DTLB. This setup is more complex than the one considered by many current analysis approaches that only consider I$ or I$+D$ but not consider the TLBs.

- multiple-level: This setup in addition to using the same first level cache memories as the previous setup, includes a L2 cache. This is a major advance with respect to the state of the art in timing analysis where very few authors have
Figure 4.1: Reference architectures. The TLBs (both instruction and data), the data cache and the L2 cache are optional and their use depend on the particular cache architecture implemented: (a) instruction cache only, (b) single-level caches or (c) multi-level caches.

Table 4.1: Duration of memory accesses (\textit{ilat} for instructions and \textit{dlat} for data) for each of the setups and hit/miss combinations in each cache level

<table>
<thead>
<tr>
<th>Setup</th>
<th>I$</th>
<th>ITLB</th>
<th>D$</th>
<th>DT LB</th>
<th>L2</th>
<th>ilat</th>
<th>dlat</th>
</tr>
</thead>
<tbody>
<tr>
<td>icache-only</td>
<td>hit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>miss</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td>single-level</td>
<td>hit</td>
<td>hit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>miss</td>
<td>hit</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>hit</td>
<td>miss</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>miss</td>
<td>-</td>
<td>-</td>
<td>-</td>
<td>200</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>hit</td>
<td>hit</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>miss</td>
<td>hit</td>
<td>-</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>hit</td>
<td>miss</td>
<td>-</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>miss</td>
<td>miss</td>
<td>-</td>
<td>200</td>
<td>-</td>
</tr>
<tr>
<td>multi-level</td>
<td>hit</td>
<td>hit</td>
<td>-</td>
<td>-</td>
<td>hit</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>miss</td>
<td>hit</td>
<td>-</td>
<td>-</td>
<td>hit</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>miss</td>
<td>hit</td>
<td>-</td>
<td>-</td>
<td>miss</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>hit</td>
<td>miss</td>
<td>-</td>
<td>-</td>
<td>hit</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>miss</td>
<td>miss</td>
<td>-</td>
<td>-</td>
<td>hit</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>hit</td>
<td>miss</td>
<td>-</td>
<td>200</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>hit</td>
<td>hit</td>
<td>-</td>
<td>1</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>miss</td>
<td>hit</td>
<td>-</td>
<td>10</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>miss</td>
<td>hit</td>
<td>miss</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>hit</td>
<td>miss</td>
<td>hit</td>
<td>100</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>miss</td>
<td>miss</td>
<td>hit</td>
<td>110</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>-</td>
<td>-</td>
<td>miss</td>
<td>miss</td>
<td>miss</td>
<td>200</td>
<td>-</td>
</tr>
</tbody>
</table>

tried to analyse several levels of cache, achieving it for very constrained configurations [19]. Our L2 cache can be configured in terms of \textit{inclusivity} and \textit{write-allocate} and \textit{write-miss} policies in different ways. For the purpose of the experiments of this section we have chosen (i) non-inclusive instruction caches and inclusive data caches whenever the L2 cache is in place, (ii) write-through non-write allocate first level caches, and (iii) write-back write allocate L2 caches.

For each of the setups Table 4.1 shows the duration of \textit{imisslat} and \textit{dmisslat}.
4.2 Benchmark Suites

We use both PROARTIS-designed synthetic and standard benchmarks for the evaluation of our analysis techniques. The former are small programs meant to better show how our technique works, while the latter are used to evaluate our techniques with more realistic programs.

**EEMBC**: As standard benchmarks, we used benchmarks from the EEMBC Autobench suite [23] as reference for the analysis of single-path programs. EEMBC Autobench is a well-known benchmark suite that reflects the current real-world demand of some embedded systems. We used: a2time (A2T), aifftr (AIFF), aifirf (AIFI), aiifft (AII), cacheb (CA), canrdr (CN), iirflt (IIR), puwmod (PU), rspeed (RS), tlbook (TB) and ttsprk (TT).

**Mälardarlen**: For multipath-program analysis we used several Mälardarlen benchmarks [16], which are commonly used in the community to evaluate and compare different types of WCET analysis tools and methods. In particular we used: bs (BS), cnt (CNT), compress (COM), crc (CRC), insertsort (INS), qsort (QSO) and select (SEL). For each of these programs we developed several input sets that exercise different paths.

**PROARTIS Synthetic Benchmarks (PSB)**: We also designed several synthetic benchmarks that helped analysing in detail the main results obtained with our analysis techniques.

**PSB-multipath**: We developed a multipath synthetic benchmark to better understand how the multipath-program time analysis works. In the synthetic benchmark we know (1) the different paths of the program, (2) the different input set that exercise each path and (3) the longest path. Thus we may compare the pWCET estimation we obtain with our single-path time analysis when applied to the worst-case path against the pWCET estimation we obtain with our multiple-path time analysis when applied to a set of paths which includes the worst-case path.

**PSB-DEC**: We develop another synthetic benchmark for the analysis of time composability aspects of PROARTIS called PSB Different Execution conditions or PSB-DEC. PSB-DEC performs \( n \) different memory access to an array, each accessing to different array positions and performed with a different instruction. The \( n \) memory accesses are within a loop with a parametrisable number of iterations. Each instruction performs an access to the instruction and data caches. The code of the benchmark and the data footprint fit in instruction and data cache respectively. As a result, the first iteration of the loop fills the instruction and data caches, which can be reused by the subsequent iterations. The higher is the number of iterations (\( \text{PROGRAM ITERATIONS} \)), the lower is the impact due to cold misses of filling the instruction and data access.

**PSB-SW**: In order to illustrate the potential and tradeoffs involved in software randomisation, we have designed a specific synthetic benchmark (PSB-SW). The synthetic benchmark consists of a loop which contains calls to four distinct functions. This structure is very similar to the one of the Automotive EEMBC benchmarks. Each of the functions receives two arguments and has a return value, while it contains local variables in order to exhibit stack usage. The code of the functions is linear, without any control flow which may increase the benefit from the instruction cache inside an iteration, and performs integer computations over its local data. No global variables or floating operations were used. Although each of the
functions has a different size, the total code size of the four functions and the main one containing the loop, equals the instruction cache size. Therefore, the random placement introduced by the compiler technique creates different cache conflicts in the instruction cache. All the experiments were performed with 100 iterations.

4.3 Icache-only Setup: Results

The SPTA technique provides a very tight approximation to the actual execution time distribution of the program. This is achieved, however, at the cost of being time consuming as it has to convolve the ETP for every instruction in the program. In addition to that, deriving the ETP of instructions in a complex architecture is also time consuming. For these reasons, MBPTA-EVT is the preferred PROARTIS analysis technique over SPTA. The latter is used to check the tightness of the former.

Overall in this section we focus on an architecture with a single on-chip piece of cache the instruction cache. This is enough to show evidence of how MBPTA-EVT behaves in the presence of cache and reduce the timing requirements to compute SPTA. For the same reason of time consumption we focus on the Evict on Access miss policy. We could use Evict on Miss which would lead to better (tighter) WCET estimations, but would require to track the hit/miss history of previous accesses to determine the hit/miss probability of each access, making it time consuming.

4.3.1 MBPTA-EVT autoevaluation

In this section we evaluate that the requirements imposed by EVT are met by our architecture. We also evaluate the sensitivity of the pWCET estimations to the different parameters explained in Section 2.3.2.

Data fits the Gumbel distribution. In order to obtain sound pWCET estimations through the application of EVT, the platform/data under consideration has to meet certain statistical properties. One of the ways to check whether this is the case is via hypothesis testing. We start with the hypothesis that our execution time observations fit the Gumbel distribution: \( H_0 : F = F_\xi \). To that end we use the ET test [13] as introduced in Section 2.3.

When applying the ET test, we check a relation that is proved equivalent to the hypothesis \( H_0 \) in [13]. This relation verifies that a certain parameter \( \hat{q}_{ET,n} \) calculable for any set of data belongs to an imposed interval. This interval is also proved calculable for any set of data. For all our data we observed that \( \hat{q}_{ET,n} \) belongs to the expected interval. We provide in Table 4.2 the results of the ET test for the EEMBC benchmarks. As shown, the \( \hat{q}_{ET,n} \) (estimated \( q \) in the table) is always within the bounds of the confidence interval (lower bound and upper bound in the table).

Independent and identically distributed observations. The application of EVT also requires that the data are i.i.d., which is ensured by a combination of hardware with suitable randomisation properties and a suitable experimental methodology when observing the executions of the program under consideration (see Section 2.3.2).
Table 4.2: Results of ET test for the icache-only configuration

<table>
<thead>
<tr>
<th>benchmark</th>
<th>estimated ( q )</th>
<th>lower bound</th>
<th>upper bound</th>
<th>Test passed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2T</td>
<td>5853724.46</td>
<td>5852834.23</td>
<td>5865506.74</td>
<td>ok</td>
</tr>
<tr>
<td>AIFF</td>
<td>23517652.36</td>
<td>23509682.03</td>
<td>23537034.87</td>
<td>ok</td>
</tr>
<tr>
<td>AIFI</td>
<td>16729538.90</td>
<td>16729128.13</td>
<td>16742329.99</td>
<td>ok</td>
</tr>
<tr>
<td>AII</td>
<td>22882675.2</td>
<td>22869204.77</td>
<td>22891866.56</td>
<td>ok</td>
</tr>
<tr>
<td>CA</td>
<td>905926.69</td>
<td>905888.07</td>
<td>911595.77</td>
<td>ok</td>
</tr>
<tr>
<td>CN</td>
<td>2326718.22</td>
<td>2323995.85</td>
<td>2331944.53</td>
<td>ok</td>
</tr>
<tr>
<td>IIR</td>
<td>35240637.10</td>
<td>35234912.17</td>
<td>35251478.74</td>
<td>ok</td>
</tr>
<tr>
<td>PU</td>
<td>3375784.90</td>
<td>3374782.92</td>
<td>3385247.76</td>
<td>ok</td>
</tr>
<tr>
<td>RS</td>
<td>773449.23</td>
<td>773359.02</td>
<td>778378.88</td>
<td>ok</td>
</tr>
<tr>
<td>TB</td>
<td>17955589.34</td>
<td>17953579.34</td>
<td>17968033.12</td>
<td>ok</td>
</tr>
<tr>
<td>TT</td>
<td>1714338.80</td>
<td>1713821.50</td>
<td>1720588.58</td>
<td>ok</td>
</tr>
</tbody>
</table>

We start by checking that the data are identically distributed.

- **Identical distribution:** We apply the two-sample Kolmogorov-Smirnov [11] test in which any two sample sets are compared. The Kolmogorov-Smirnov (KS) statistic quantifies a distance between the empirical distribution functions of two samples. The null hypothesis \( H_0 \) is that the both samples are identically distributed and the alternative hypothesis \( H_1 \) is that the samples are not identically distributed. For our experiments we use a significance level \( \alpha = 0.05 \), which is a common choice in this type of test. The outcome of the test is called the p-value. If the p-value is higher than \( \alpha \) the null hypothesis cannot be rejected, which means that both samples are identically distributed.

- **Independence:** In order to prove that samples are independent we use the runs test for randomness [4]. This test is used to check whether a series of binary events can be considered to be randomly distributed, and hence independent. In this test, the null hypothesis is that the data are randomly distributed (independent) and the alternative hypothesis is that the data are not randomly distributed (not independent). A run is defined as a sequence of identical events, preceded or succeeded by different events or no events. The runs test used here applies to binomial distributions only. For example, in a sequence 0110111, we have 4 runs (0, 11, 0, 111). Our data are continuous, hence a cut-point must be chosen by the user so that the data are transformed into a binary sample. The binary transformation used consists of comparing each execution time with the median across all runs, so that the execution time is replaced by 0 if lower than the median and by 1 otherwise. The expectation of the number of runs \( R \) is given by: \( \mathbb{E}(R) = 2mn/N \), where \( m \) is the number of events of type 1, and \( n \) the number of events of type 2, and \( N \) is the total sample size. The variance of the number of runs \( R \) is given by \( \mathbb{V}(R) = \frac{2mn(2mn-N)}{N^2(N-1)} \).

Let \( r \) be the number of runs measured in the sample. It is known that when \( m \) or \( n \) tend to infinity then \( Z = \frac{r - \mathbb{E}(R)}{\sqrt{\mathbb{V}(R)}} \rightarrow N(0,1) \), where \( N(0,1) \) is the standard normal distribution [4]. This means that the test statistic, \( Z \), is asymptotically normally distributed. From the value of \( Z \), we compute the p-value with a significance level \( \alpha = 5\% \). If the p-value is higher than alpha the null hypothesis cannot be rejected, which means that data are randomly distributed.

The last column in Table 4.3 shows the results of the i.i.d. tests. The second and
third columns show the p-value obtained by applying the KS test to the execution times obtained for a sample of 10,000 observations. From this original sample we create two smaller samples of $m$ elements. For this experiment we use two values for $m$: 50 and 100. We populate the smaller samples by randomly taking elements from the original sample, which ensures that the smaller samples maintain the same statistical properties as the original [11]. In our case, the property of interest is the distribution. Then we apply the two-sample KS test to the two smaller samples. We observe that in all cases the p-value is higher than the significance level (0.05 as explained before) which indicates that data are identically distributed. The fourth column in Table 4.3 shows the p-value computed for each EEMBC benchmark with 10,000 runs for the independence test. We observe that for all the benchmarks under consideration the runs we collect from them are independent.

**Minimum Number of runs (observations).** Figure 4.2 shows the variation of CRPS as we add more runs using $N_{\text{delta}} = 50$ and as initial value for $N_{\text{current}} = 50$. The smaller these values are the more precise selection of the minimum number of runs ($mnr$) we can do, but at the cost of doing more EVT-projections. We found that these two values represent a good balance. We have split the results in Figure 4.2 into 3 charts for the sake of clarity. Each of the charts groups a subset of the EEMBC benchmarks.

In order to obtain the minimum number of observations we use the algorithm presented in Section 2, which focuses on the convergence of the EVT tail projection as more runs are considered, taking CRPS as the convergence metric. We set the threshold to 0.01 and $N_{\text{delta}}$ to 50. We observe that although CRPS decreases, it is not monotonically decreasing. The case of the aifirf benchmark is shown in Figure 4.3. As shown, CRPS decreases until 200 runs, then increases until 300 runs before decreasing again, although it always decreases in the long run. For this reason our convergence algorithm takes into account small variations. Table 4.4 shows the minimum number of runs required for each EEMBC benchmarks. On average we need 404 runs per benchmark. The average time required to run an EEMBC benchmark in our simulation infrastructure is about 1 minute, so the total time required was $404 \times 11/60 = 74$ hours to run all benchmarks. This is equivalent to about one hour of simulations in a 64-node cluster, which is a fairly common asset in industrial production systems. Note that final users will run their applications on real hardware, so performing some hundreds of runs of an application whose
Figure 4.2: Benchmark CRPS variation as we increase number of runs. $N_{\text{delta}} = 50$ and threshold = 0.1 for the icache-only configuration.

Figure 4.3: Minimum number of observations for the aifirf Autobench benchmark.

evolution time is in the order of milliseconds will take few seconds. It is worth noting that for building Figure 4.2 we used 700 runs, which provides satisfactory convergence for a threshold of $10^{-2}$. However, to achieve convergence at lower CRPS values, e.g. $10^{-3}$ we would need to increase the number of observations.

**pWCET estimations for single-path programs and comparison with SPTA.**

Figure 4.4 shows the pWCET estimations provided by MBPTA and by SPTA for
Table 4.4: Minimum number of observations per EEMBC Autobench benchmark.

<table>
<thead>
<tr>
<th></th>
<th>A2T</th>
<th>AIFF</th>
<th>AIFI</th>
<th>AIH</th>
<th>CA</th>
<th>CN</th>
<th>HR</th>
<th>PU</th>
<th>RS</th>
<th>TB</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>500</td>
<td>300</td>
<td>300</td>
<td>450</td>
<td>350</td>
<td>450</td>
<td>300</td>
<td>550</td>
<td>600</td>
<td>300</td>
<td>350</td>
</tr>
</tbody>
</table>

Figure 4.4: pWCET estimations for the aifirf and rspeed benchmarks for icache-only configuration.

Table 4.5: MBPTA-EVT pWCET estimations for EEMBC benchmarks w.r.t. SPTA

<table>
<thead>
<tr>
<th>benchmark</th>
<th>Absolute SPTA (10^{-13})</th>
<th>Relative MBPTA-EVT (10^{-13})</th>
<th>Relative MBPTA-EVT (10^{-16})</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2T</td>
<td>5946502</td>
<td>1.1%</td>
<td>2.0%</td>
</tr>
<tr>
<td>AIFF</td>
<td>23687926</td>
<td>1.2%</td>
<td>1.6%</td>
</tr>
<tr>
<td>AIFI</td>
<td>16820831</td>
<td>0.7%</td>
<td>0.9%</td>
</tr>
<tr>
<td>AIH</td>
<td>23051707</td>
<td>0.5%</td>
<td>1.0%</td>
</tr>
<tr>
<td>CA</td>
<td>939142</td>
<td>4.3%</td>
<td>6.3%</td>
</tr>
<tr>
<td>CN</td>
<td>2384692</td>
<td>2.4%</td>
<td>3.5%</td>
</tr>
<tr>
<td>HR</td>
<td>35387912</td>
<td>0.3%</td>
<td>0.5%</td>
</tr>
<tr>
<td>PU</td>
<td>3447660</td>
<td>2.0%</td>
<td>3.1%</td>
</tr>
<tr>
<td>RS</td>
<td>810337</td>
<td>4.5%</td>
<td>7.0%</td>
</tr>
<tr>
<td>TB</td>
<td>18064300</td>
<td>0.7%</td>
<td>1.0%</td>
</tr>
<tr>
<td>TT</td>
<td>1762839</td>
<td>3.6%</td>
<td>5.3%</td>
</tr>
</tbody>
</table>

two EEMBC benchmarks, rspeed and aifirf (due to space constraints we cannot show pWCET estimations graphically for all benchmarks). The x-axis shows the pWCET estimation and the y-axis shows the associated probabilities. We can observe that the difference between the pWCET with SPTA and with MBPTA-EVT is really low. Table 4.5 shows the pWCET estimations obtained with MBPTA-EVT for different target probabilities with respect to the estimations obtained with SPTA. We observe that the estimations for \(10^{-13}\) are always below 5% with an average of 2%. For \(10^{-16}\) the difference is always below 7% and the average difference is 3%.

Effect of the block size when applying block maxima. The size of the blocks determines the portion of the original distribution that is considered the tail. The larger the block size, the better the fit to the tail; however, increasing the
block size results in fewer blocks from a fixed trace and thus fewer values in the final sample. In the extreme case, if a single block is used, we will have only one block maximum value corresponding to the maximum of the original distribution. Conversely, if we use as many blocks as elements in the original distribution, the distribution of the block maximum values will gather the entirety of the original distribution, rather than capturing the essence of the tail. An optimal block size does not exist. Small blocks increase tightness for high exceedance probabilities at the expense of pessimism for lower probabilities. Conversely, large blocks improve accuracy for lower exceedance probabilities. At this stage of the project, the block size is qualitatively chosen around 50 (observations per block), which shows to be very tight in the range of probabilities of interest.

**pWCET estimations for multiple-path programs and comparison with SPTA.** In this section we show the pWCET estimations provided for the PSB2. For MBPTA we vary the number of runs for the worst path and for the non-worst paths. We label each estimation as x-y where x is the number of runs we consider of non worst paths and y the number of runs from the worst case path. For instance, 0-10000 is the pWCET estimation when we consider 10,000 runs of the worst path and 9000-1000 is the pWCET estimation when we consider 9,000 runs of non worst-case paths and 1,000 runs of the worst-case path. For every path we make at least the minimum number of observations.

In Figure 4.6 the x-axis shows the pWCET estimation and the y-axis shows the associated probabilities. We observe that all MBPTA-EVT estimations safely upperbound the SPTA pWCET estimation. Only for the case 10000-0 we observe that the estimation is not safe. This case illustrates our claim in Section 2: pWCET estimates for untested paths cannot be made, as the execution times for those paths would no longer be drawn from an identical distribution. The other EVT estimations that consider the worst path are all an upper bound for the SPTA and have a small variation between them, which indicates that the frequency at which the worst-case path is exercised with the given input data set affects the provided WCET estimation only slightly.
Figure 4.6: pWCET estimations for the synthetic benchmark for icache-only configuration.

Table 4.6: Result of the i.i.d. test for the icache-only configuration

<table>
<thead>
<tr>
<th>benchmark</th>
<th>S=50</th>
<th>S=100</th>
<th>p</th>
<th>Both test passed?</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2T</td>
<td>0.65</td>
<td>0.41</td>
<td>0.42</td>
<td>ok</td>
</tr>
<tr>
<td>AIFF</td>
<td>0.36</td>
<td>0.32</td>
<td>0.57</td>
<td>ok</td>
</tr>
<tr>
<td>AIFI</td>
<td>0.30</td>
<td>0.47</td>
<td>0.40</td>
<td>ok</td>
</tr>
<tr>
<td>AH</td>
<td>0.43</td>
<td>0.48</td>
<td>0.43</td>
<td>ok</td>
</tr>
<tr>
<td>CA</td>
<td>0.51</td>
<td>0.69</td>
<td>0.58</td>
<td>ok</td>
</tr>
<tr>
<td>CN</td>
<td>0.26</td>
<td>0.39</td>
<td>0.39</td>
<td>ok</td>
</tr>
<tr>
<td>IIR</td>
<td>0.44</td>
<td>0.35</td>
<td>0.53</td>
<td>ok</td>
</tr>
<tr>
<td>PU</td>
<td>0.57</td>
<td>0.78</td>
<td>0.41</td>
<td>ok</td>
</tr>
<tr>
<td>RS</td>
<td>0.23</td>
<td>0.26</td>
<td>0.30</td>
<td>ok</td>
</tr>
<tr>
<td>TB</td>
<td>0.18</td>
<td>0.47</td>
<td>0.45</td>
<td>ok</td>
</tr>
<tr>
<td>TT</td>
<td>0.60</td>
<td>0.42</td>
<td>0.54</td>
<td>ok</td>
</tr>
</tbody>
</table>

4.4 Single-level cache setup: Results

This setup features all first level cache memories: I$, D$, ITLB and DTLB. This setup is more complex than the one considered by many current analysis approaches that only consider I$ or I$+D$ but not consider the TLBs.

4.4.1 Single-Path Programs

We start by showing that this setup also accomplishes the requirements imposed by EVT in term of i.i.d. execution times of EEMBC. EEMBC are single-path program that only exploit part of our MBPTA-EVT technology. In the next section, we focus on multi-path programs. Table 4.6 show that all tests are passed.

By comparing Table 4.7 and Table 4.4 we observe that the effect of adding data cache and ITLB and DTLB has no big effect on the minimum number of runs required. In all cases the minimum number of runs stay well below 1000. Noticeably, in this second case the threshold defined is satisfied since the first step, hence the algorithm concludes about the minimum number of runs/observations after 5 steps.

Table 4.8 show the WCET estimations we obtain for probabilities $10^{-10}$, $10^{-13}$, $10^{-16}$; the values are normalised with respect to those obtained for a probability $10^{-10}$.
Technical Deliverables D3.4 for Milestone MS2  
Version 1.0

Table 4.7: Minimum number of observations per EEMBC Autobench benchmark

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>A2T</th>
<th>AIFF</th>
<th>AIFI</th>
<th>AII</th>
<th>CA</th>
<th>CN</th>
<th>IIR</th>
<th>PU</th>
<th>RS</th>
<th>TB</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Observations</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
<td>300</td>
</tr>
</tbody>
</table>

Table 4.8: MBPTA pWCET estimations for EEMBC benchmarks for the single-level configuration

<table>
<thead>
<tr>
<th>Benchmark</th>
<th>Absolute pWCET $10^{-10}$</th>
<th>Relative MBPTA-EVT $10^{-13}$</th>
<th>Relative MBPTA-EVT $10^{-16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2T</td>
<td>2276850</td>
<td>21.6%</td>
<td>43.3%</td>
</tr>
<tr>
<td>AIFF</td>
<td>10950331</td>
<td>19.8%</td>
<td>39.6%</td>
</tr>
<tr>
<td>AIFI</td>
<td>3165602</td>
<td>13.2%</td>
<td>26.4%</td>
</tr>
<tr>
<td>AII</td>
<td>10482026</td>
<td>2.4%</td>
<td>6.2%</td>
</tr>
<tr>
<td>CA</td>
<td>221107</td>
<td>3.2%</td>
<td>7.3%</td>
</tr>
<tr>
<td>CN</td>
<td>668974</td>
<td>4.9%</td>
<td>9.6%</td>
</tr>
<tr>
<td>IIR</td>
<td>12043575</td>
<td>19.1%</td>
<td>39.5%</td>
</tr>
<tr>
<td>PU</td>
<td>934077</td>
<td>3.0%</td>
<td>6.0%</td>
</tr>
<tr>
<td>RS</td>
<td>206178</td>
<td>2.0%</td>
<td>4.3%</td>
</tr>
<tr>
<td>TB</td>
<td>3391076</td>
<td>16.4%</td>
<td>31.6%</td>
</tr>
<tr>
<td>TT</td>
<td>469646</td>
<td>4.4%</td>
<td>9.1%</td>
</tr>
</tbody>
</table>

We observe that the WCET estimations vary significantly as we decrease the target probability. This is so because few more misses due to random events have a large relative impact in execution time, thus making distribution wide and, therefore, the slope of the pWCET estimation is low. As shown later, this effect is largely mitigated when L2 caches are in place because they drastically decrease the impact of those misses in execution time by reducing the number of accesses that must reach slow main memory.

4.4.2 Multiple-Path Programs

As mentioned in the Analysis section, the application of EVT MBPTA to multiple-path programs is simple in principle as long as the compound distribution resulting from multiple paths must exhibit the required i.i.d. properties and a sufficient number of observations must be made for each path. It is also important to note that the result only applies to the set of observed paths.

**pWCET estimations for the synthetic benchmark.** In order to check the results EVT-MBPTA obtains for multipath programs we use the synthetic benchmark presented in Section 4.2. This is a multi-path program for which we know the worst-case path, which is significantly longer than any other path. As a reference comparison point we take the pWCET estimation we obtain with our single-path technique when applied to the worst path. We compare it with the pWCET estimations we obtain when apply the multi-path technique to different sets of paths in which we include the worst path. For all the experiments in this section, we also applied the test for independence and identical distribution described in the previously.

Figure 4.7 shows the pWCET estimation obtained with MBPTA-EVT for PSB2 when varying the number of runs for the worst path and for the non-worst paths. We label each estimation as $x-y$ where $x$ is the number (in thousands) of runs we consider of non worst paths and $y$ the number of runs from the worst case path.
Technical Deliverables D3.4 for Milestone MS2
Version 1.0

Figure 4.7: pWCET estimations for the synthetic benchmark for the single-level configuration

For instance, 0-10 is the pWCET estimation when we consider 10,000 runs of the worst path and 9-1 is the pWCET estimation when we consider 9,000 runs of non-worst-case paths and 1,000 runs of the worst-case path. For every path we make at least the minimum number of observations.

In Figure 4.7 we compare all estimations obtained with MBPTA-EVT with the result we obtain with SPTA for the worst-case path (SPTA \textit{wc}). First, and most importantly, we observe that all multi-path estimations that consider the worst path are an upper bound for the SPTA. In the particular case of the 10-0 estimation, we consider no runs from the worst path in the EVT. This case illustrates our claim in Section 2: pWCET estimates for untested paths cannot be made, as the execution times for those paths would no longer be drawn from an identical distribution. We observe that all multi-path estimations that consider the worst path are quite close each other. Only, in the particular case of the 10-0 estimation, when we consider no runs from the worst path in the EVT we observe a significant variation in the WCET estimation.

**pWCET estimations for standard benchmarks.** We compare the pWCET estimations we obtain for different Malardalen WCET benchmarks. As in the previous experiments, the data passed all Gumbel fitting (not shown) and i.i.d. tests (see Table 4.9).

We compare the pWCET estimation when we consider 10,000 runs from the worst-case path (wc) against the case in which we consider 10,000 runs from all paths including the worst case path (all). Figure 4.8 shows the results for the BS, CNT, COMPRESS and CRC benchmark. Again we observe that as soon as the worst-case path is considered as part of the data set, EVT provides results comparable to considering only the worst-case path. The pessimism of the latter over the former, is typically around 10% for the benchmarks we have tested.

### 4.4.3 Cache configuration sensitivity

We also study how sensitive MBPTA-EVT projections are to the cache configurations used. For that purpose we vary associativity from 256-way (fully-associative) down to 1-way (direct-mapped) including 32-way and 8-way configurations.

Table 4.10 shows the pWCET increment of all cache configurations with respect to the fully-associative one for all EEMBC benchmarks when considering an ex-
ceedance probability of $10^{-13}$. As expected, the fully-associative cache provides the lowest pWCET estimations. That is, as pointed in D1.2, the random replacement policy has lower probability of resulting in cache layouts with multiple cache conflicts. However, as we reduce the associativity of the cache, and so we increase the number of sets, the number of cache layouts decreases, thus increasing the probability of having more cache conflicts.

The pWCET increment due to reducing the cache associativity depends on the application. For instance, a2time is very sensitive to cache associativity which makes the pWCET estimations to increase up to $6 \times$ when considering set-associative caches and up to $19 \times$ when considering a direct-mapped cache, with respect to the fully-associative cache. Instead, in the case of aifftr, we observe pWCET estimations increments of only 9% and 11% when considering set-associative caches and up to $6 \times$ when considering a direct-mapped cache, with respect to the fully-associative cache.

### 4.5 Multiple-level cache setup: Results

#### 4.5.1 Single-path programs

The final setup that we have called multi-level cache setup, represents a processor architecture extremely hard to analyse with static analysis. In fact, the authors of this document only know two papers in which it has been provided WCET estimations with Static Timing Analysis tools for a processor equipped with several levels of cache. The source of the problem lies in the fact that STA requires to keep all the cache state at every single point in time. The difficulty of doing this with two levels of caches is significantly higher than with one since it has to be tracked when we have a miss in the lower cache levels (the one closer to the processor).
Figure 4.8: pWCET estimations with MBPTA-EVT when applied to the known worst-path (only-wc) only and to a set of paths including the worse (all) for the single-level cache architecture.

to determine when the higher levels are accessed. Moreover, none of the previous papers consider simultaneously several levels of cache and TLBs.

With MBPTA-EVT it has to be ensured that the architecture accomplishes with EVT requirements. It has been shown in D1.2 how this can be done following a bottom up approach, starting from processor resources (blocks) that are combined to conform the processor architecture. The way in which these blocks are connected can be arbitrarily complex and the number of devices arbitrarily long.

i.i.d. tests and Gumbel fitting. As with the previous setups, all the benchmarks run on this setup successfully passed all the independence and identical distribution tests. The tail of the data shown to fit the Gumbel distribution.

Minimum number of runs/observations.
Table 4.11 shows how our algorithm requires at most 1000 observations in order to conclude about the statistical relevance of the input MBPTA distribution.
Table 4.10: pWCET increment of the set associativity and direct map caches with respect to the full associativity cache, considering an exceedance probability of $10^{-13}$

<table>
<thead>
<tr>
<th>Benchmarks</th>
<th>32 ways 8 sets</th>
<th>8 ways 32 sets</th>
<th>1 way 256 sets</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2T</td>
<td>452%</td>
<td>511%</td>
<td>1758%</td>
</tr>
<tr>
<td>AIFF</td>
<td>9%</td>
<td>11%</td>
<td>468%</td>
</tr>
<tr>
<td>AIFI</td>
<td>61%</td>
<td>65%</td>
<td>1418%</td>
</tr>
<tr>
<td>AII</td>
<td>9%</td>
<td>12%</td>
<td>653%</td>
</tr>
<tr>
<td>CA</td>
<td>9%</td>
<td>12%</td>
<td>904%</td>
</tr>
<tr>
<td>CN</td>
<td>8%</td>
<td>9%</td>
<td>2126%</td>
</tr>
<tr>
<td>IIR</td>
<td>370%</td>
<td>478%</td>
<td>1448%</td>
</tr>
<tr>
<td>PU</td>
<td>29%</td>
<td>31%</td>
<td>855%</td>
</tr>
<tr>
<td>RS</td>
<td>23%</td>
<td>25%</td>
<td>12691%</td>
</tr>
<tr>
<td>TB</td>
<td>167%</td>
<td>185%</td>
<td>1995%</td>
</tr>
<tr>
<td>TT</td>
<td>13%</td>
<td>14%</td>
<td>3386%</td>
</tr>
</tbody>
</table>

Table 4.11: Minimum number of observations per EEMBC Autobench benchmark - L1L2TLB

<table>
<thead>
<tr>
<th></th>
<th>A2T</th>
<th>AIFF</th>
<th>AIFI</th>
<th>AII</th>
<th>CA</th>
<th>CN</th>
<th>IIR</th>
<th>PU</th>
<th>RS</th>
<th>TB</th>
<th>TT</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>700</td>
<td>600</td>
<td>1000</td>
<td>550</td>
<td>650</td>
<td>400</td>
<td>450</td>
<td>300</td>
<td>400</td>
<td>600</td>
<td>350</td>
</tr>
</tbody>
</table>

Effect of performance-improving hardware on pWCET estimations. It is acknowledged that a cache structure with several levels of cache can significantly boost performance. However, there is few data supporting the potential benefit of that cache structure may have on WCET estimation. In real-time systems the average performance is important as it affects key metrics like energy consumption, but the main focus is on reducing (tightening) the WCET estimation keeping it safe.

In this section we show how the pWCET estimations obtained with MBPTA-EVT take significant benefit of a multi-level cache structure. Figure 4.9 shows the pWCET estimations we obtain with the puwmod01 benchmark. We observe that for the range of probabilities under consideration ($10^{-13} - 10^{-16}$) this difference is around 20%.

Table 4.12 shows the results for all EEMBC benchmarks. In particular it shows the reduction in pWCET for each target probability under consideration. For example the second column shows $\frac{pWCET_{\text{single-level}}}{pWCET_{\text{multi-level}}}$. We observe that for some benchmarks the reduction in pWCET is more than $10\times$ while the average reduction in pWCET for all benchmarks and all three target probabilities is $3\times$.

4.5.2 multi-path programs

In this section we show some counter charts and tables for those presented for multi-path programs under the single-level cache architecture. This is done for completeness reasons only, since the main conclusions we obtain there also apply here.
Figure 4.9: Single-path pWCET estimation for one benchmark (puwmod) with two different configurations (L1TLB and L1L2TLB)

Table 4.12: MBPTA pWCET estimations for EEMBC benchmarks the multi-level setup normalised to pWCET estimation for the single-level setup for the same target probability.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>$10^{-10}$</th>
<th>$10^{-13}$</th>
<th>$10^{-16}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>A2T</td>
<td>5.9</td>
<td>6.8</td>
<td>7.7</td>
</tr>
<tr>
<td>AIFF</td>
<td>2.2</td>
<td>2.2</td>
<td>2.2</td>
</tr>
<tr>
<td>AIFI</td>
<td>2.5</td>
<td>2.8</td>
<td>3.0</td>
</tr>
<tr>
<td>AH</td>
<td>2.2</td>
<td>2.3</td>
<td>2.3</td>
</tr>
<tr>
<td>CA</td>
<td>1.3</td>
<td>1.3</td>
<td>1.3</td>
</tr>
<tr>
<td>CN</td>
<td>1.3</td>
<td>1.3</td>
<td>1.4</td>
</tr>
<tr>
<td>IIR</td>
<td>8.1</td>
<td>9.1</td>
<td>10.2</td>
</tr>
<tr>
<td>PU</td>
<td>1.2</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>RS</td>
<td>1.1</td>
<td>1.2</td>
<td>1.2</td>
</tr>
<tr>
<td>TB</td>
<td>3.1</td>
<td>3.5</td>
<td>3.8</td>
</tr>
<tr>
<td>TT</td>
<td>1.2</td>
<td>1.3</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Figure 4.10: pWCET estimations for the synthetic benchmark for the multi-level configuration
Figure 4.11: pWCET estimations with MBPTA-EVT when applied to the known worst-path (only-wc) only and to a set of paths including the worse (all) for the multi-level cache architecture

4.6 Comparison with Static Timing Analysis and deterministic architectures

In this section we compare the average performance difference between a deterministic cache hierarchy and a randomised one. We also provide an initial comparison of how MBPTA-EVT behaves in comparison with Static Timing Analysis.

In this section we use the two cache configurations (single-level and multi-level) that we have mostly used so far. We implemented a variant of those cache architectures in which the associativity and size remain the same for all five cache memories: level one instruction and data caches, instructions and data TLBs and the L2 cache. The only change we introduce in their functioning is that we implement a modulo placement and LRU replacement instead of our random placement and replacement policies.

Average performance. We compare the performance we obtain when we run
EEMBC benchmarks on modulo+lru based cache setups and random-based cache setups.

In Figure 4.12 we compare the performance obtained with modulo+lru, represented with the dashed vertical line, and the distribution of execution times we obtain with random caches. We focus on the a2time and aiifft benchmarks.

In all four charts in Figure 4.12 we observe that most of the observed execution times with the randomised cache architectures are higher than the execution time with the deterministic cache architectures. In some cases the observations are more than 3×. However, average values are much more close. The difference between the performance of deterministic caches and the average performance of the randomised ones is 1.25, 1.03, 1.04 and 1.02 respectively for Figures Figure 4.12(a), Figure 4.12(b), Figure 4.12(c) and Figure 4.12(d) respectively.

Table 4.13 shows the average performance improvement of the deterministic cache over the randomised ones for both setups single-level and multi-level. The max-
Table 4.13: Average performance in execution cycles for EEMBC benchmarks.

<table>
<thead>
<tr>
<th>benchmark</th>
<th>deterministic cache</th>
<th>randomised cache</th>
<th>Randomised cache slowdown</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>single-level</td>
<td>multi-level</td>
<td>single-level</td>
</tr>
<tr>
<td>A2T</td>
<td>299959</td>
<td>270329</td>
<td>612221</td>
</tr>
<tr>
<td>AIFF</td>
<td>9895465</td>
<td>4525633</td>
<td>20009331</td>
</tr>
<tr>
<td>AIFI</td>
<td>1234805</td>
<td>1073472</td>
<td>1110394</td>
</tr>
<tr>
<td>AII</td>
<td>8993927</td>
<td>4315847</td>
<td>9318243</td>
</tr>
<tr>
<td>CA</td>
<td>170797</td>
<td>155644</td>
<td>190594</td>
</tr>
<tr>
<td>CN</td>
<td>529162</td>
<td>498721</td>
<td>554833</td>
</tr>
<tr>
<td>IIR</td>
<td>1520545</td>
<td>1008087</td>
<td>3331341</td>
</tr>
<tr>
<td>PU</td>
<td>812107</td>
<td>766227</td>
<td>855555</td>
</tr>
<tr>
<td>RS</td>
<td>182806</td>
<td>174244</td>
<td>189690</td>
</tr>
<tr>
<td>TB</td>
<td>1036859</td>
<td>855259</td>
<td>1485017</td>
</tr>
<tr>
<td>TT</td>
<td>378989</td>
<td>376290</td>
<td>395783</td>
</tr>
</tbody>
</table>

The minimum slowdown experienced by randomised caches with respect to deterministic ones is 2.19X for single-level caches and 1.47X for multi-level ones. The average slowdown is 1.26X for single-level caches and 1.09X for multi-level ones. In general we observe that the randomised caches are quite competitive in terms of average performance.

**WCET comparison under different percentages of known-information at analysis time.** Making a fair comparison between two timing analysis techniques is difficult. There are many factors that affect the accuracy of the provided WCET estimations in each case. As an illustrative example, in this section we focus on the timing analysis of the cache only. To that end, we propose an evaluation framework in which the cache is the only source of WCET estimation inaccuracy. In order to prevent the processor pipeline affecting timing variability, which can be significant if the processor suffers from timing anomalies or domino effects, we assume a simple in-order pipeline in which each instruction that is not a memory operation takes a fixed latency which is known a-priori. Analogously, in order to discount the effect of other phases of the analysis, such as path analysis or loop bound analysis, we take programs with one only possible path, i.e., the input data is fixed so that all loop bounds are fixed and any branches are chosen deterministically.

Conventional Static Timing Analysis of the cache distinguishes between certain cache hits/misses and unclassified accesses. An exact static data cache analysis requires knowledge of the target addresses of all memory access operations, which is often not possible to calculate statically. As a reference configuration in our experimental environment, we assume that the cache is the only source of inaccuracy for conventional WCET estimations. We further assume that all addresses are known, so in this configuration we have the tightest WCET estimation possible, which is, in fact, the actual WCET of the program.

In addition to the idealised case in which all the information required by the analysis is available, we show how different WCET techniques react when part of this information is missing. In particular, for the conventional analysis we assume that a given percentage of the accesses to memory have an unknown address. For the conventional WCET analysis and in our experimental set-up this lack of information translates into assuming that unknown accesses are misses and that in the cache state kept by the conventional analysis, every unknown address that accesses
the cache moves all the data one position towards the LRU position. In Figure 4.13 we observe the STA rapidly increases its pessimism when a small percentage of the addresses are missing. When the amount of addresses missing is as small as 0.5% the pWCET estimations provided by MBPTA-EVT are tighter than the best WCET estimations that STA could achieve based on our model where everything is perfectly predicted but unknown addresses, which force a cache flush from the STA point of view. We also observe that in many cases the pessimism introduced by the WCET lower bound for STA increases almost linearly with the fraction of unknown accesses in the 0.5%-5% range.

4.7 Probabilistic execution time bounds under different execution conditions

We have designed an experiment to illustrate how time-composability can be considered in PROARTIS beyond considering always the worst-case initial conditions. Such experiment consists of a synthetic benchmark (PSB-DEC) which runs twice with some disturbing code run in-between. As explained before, the PSB-DEC consists of a loop whose inner code traverses an array fitting in the L1 data cache with linear code that fits in the L1 instruction cache. Therefore, a single iteration
of the loop roughly fills the cache if it is empty (first execution of the PSB-DEC), and reuses its contents if it is full (second execution of the PSB-DEC). For this experiment we consider the multi-level cache configuration since larger caches (L2 cache in this case) have higher survivability as shown in Section 3, so the potential of different execution conditions can be better explored.

Our experiments evaluate the impact in average execution time and pWCET of the second execution of the PSB-DEC (PSB-DEC\textsubscript{2}) based on the number of iterations run, which impacts the relative impact of the cache contents reuse, and the number of cache accesses of the disturbing code, which limits the amount of useful contents that the PSB-DEC\textsubscript{2} can reuse from caches.

In particular, we consider the PSB-DEC\textsubscript{2} execution with 5, 25, 50 and 100 iterations. The disturbing code accesses 0, 50, 100, 500, 1000 and 5000 unique addresses (half for data and half for instructions). Disturbing code with $M$ accesses ($\frac{M}{2}$ for data and $\frac{M}{2}$ for instructions) bounds the effect that any program with $\frac{M}{2}$ instructions may have on cache for the PSB-DEC\textsubscript{2} execution. 1000 runs have been performed for each combination.

Table 4.14 shows that experiments for all combinations of disturbing code accesses and PSB-DEC\textsubscript{2} iterations pass the i.i.d. tests.

Table 4.15 shows the average speedup of PSB-DEC\textsubscript{2} with respect to the first exe-
Table 4.15: Average speedup of PSB-DEC\textsubscript{2} with respect to PSB-DEC\textsubscript{1}

<table>
<thead>
<tr>
<th>Disturbing code size</th>
<th>PSB-DEC\textsubscript{2} iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>3.60</td>
</tr>
<tr>
<td>50</td>
<td>3.32</td>
</tr>
<tr>
<td>100</td>
<td>3.12</td>
</tr>
<tr>
<td>500</td>
<td>2.30</td>
</tr>
<tr>
<td>1000</td>
<td>1.88</td>
</tr>
<tr>
<td>5000</td>
<td>1.15</td>
</tr>
</tbody>
</table>

explanation for each combination of number of iterations and disturbing code size. As explained in Section 3, the relative impact of the cache contents reuse is inversely proportional to the size of the program being analysed. Therefore, the lower the number of iterations of PSB-DEC\textsubscript{2}, the higher the relative impact in execution time of the cache reuse. Analogously, the larger the size of the disturbing code, the lower the impact (both absolute and relative) of the cache reuse because cache contents age rapidly and survivability decreases. For instance, if the PSB-DEC executes 100 iterations (some thousands of instructions) and the disturbing code executes also some thousands of instructions (e.g., 5000), the average speedup becomes negligible.

Analogously to the average speedup, we have also computed the pWCET speedup with respect to the pWCET obtained with an empty cache. Results for $10^{-13}$ exceedance probability are shown in Table 4.16 for all combinations of number of iterations and disturbing code size. We also show the pWCET estimation for 100 iterations and different disturbing code sizes in Figure 4.14. In all cases the minimum number of runs ranges between 600 and 1000. In general, we observe that the pWCET speedup with respect to PSB-DEC\textsubscript{1} is lower than the average speedup. This is so because PSB-DEC\textsubscript{1} distribution is narrower and thus, MBPTA-EVT generates a very tight estimation with a very vertical slope in the ICDF. Conversely, PSB-DEC\textsubscript{2} shows both lower execution times and higher variability. Thus, although the MBPTA-EVT is below the one for the first run, its slope is lower and hence the gap with respect to the first run slightly decreases as the exceedance probability also decreases. This fact can be observed in the figure when comparing the case of a disturbing code with 5000 accesses with respect to the first run. In fact, even if the case of the disturbing code is a bit better on average, its pWCET becomes slightly worse than the one for the first run below a given exceedance probability. Such difference is unnecessary pessimism, but it is anyway low. Moreover, if the pWCET for a given disturbing code size is worse than the one for the empty cache (first run), the user can use the pWCET for the first run because both are equally safe and using the tightest one is better.

Overall, if the UoC executes some thousands of instructions and the code run between consecutive executions runs also some thousands of instructions, the pWCET assuming the worst initial conditions is a very tight estimation and there is no need for running further experiments.
Table 4.16: pWCET speedup of PSB-DEC\textsubscript{2} with respect to PSB-DEC\textsubscript{1} for $10^{-13}$ exceedance probability

<table>
<thead>
<tr>
<th>Disturbing code size</th>
<th>PSB-DEC iterations</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>5</td>
</tr>
<tr>
<td>0</td>
<td>2.43</td>
</tr>
<tr>
<td>50</td>
<td>2.39</td>
</tr>
<tr>
<td>100</td>
<td>1.85</td>
</tr>
<tr>
<td>500</td>
<td>1.33</td>
</tr>
<tr>
<td>1000</td>
<td>1.15</td>
</tr>
<tr>
<td>5000</td>
<td>1.02</td>
</tr>
</tbody>
</table>

Figure 4.14: pWCET estimations with MBPTA-EVT for the PSB-DEC\textsubscript{2} for 100 iterations

4.8 HyPTA: Rapita Systems Demonstrator

This section describes the experimental set-up and preliminary results for using HyPTA, implemented in RapiTime, applied to the testing of a flight guidance system for a missile.

The example missile flight system was chosen as a basis for tests. Its structure is close to the structure of IMA, but perhaps on a smaller scale. The system does not require an operating system and it is not multi-tasking. It is available with a set of tests that exercise various parts of the code thoroughly (but not necessarily along their worst case path). Therefore this example makes a good testbench for investigation of the probabilistic timing analysis of a complex application with multiple paths.

Within the system, there are 25 tasks, which are executed one immediately following the next in every cycle. The code is driven by a test harness which updates sensor inputs and runs a number of cycles. An outline of the test harness is shown in Figure 4.15.

We present the results obtained on the test harness of the missile guidance simulator for all steps of HyPTA (see description of steps in Section 2.4).

The purpose of presenting these results is to demonstrate that the data collected
during these steps is sufficient to apply MBPTA to the source code block level. We remind the reader that in Section 2.4 we presented the assumptions of applying HyPTA and these assumptions ensure the fulfilment of the hypotheses of i.i.d. required to apply MBPTA. Moreover by construction these assumptions are verified for the study we have done on the test harness.

### 4.8.1 Results of HyPTA on the missile guidance simulator

We present in Figure 4.16 the preliminary results obtained for the test harness. The results obtained by applying HyPTA are represented in green in the figure. These results are obtained from 450,000 runs at the source code block level. Based on the conclusions of the previous benchmarks presented in this section, we may conclude that there is sufficient data to apply EVT-based MBPTA at the block level.

In order to have an idea of how HyPTA behaves, in Figure 4.16 we compare the end-to-end measurement run results (red) with the WCET estimate provided by RapiTime (pink), HyPTA (green), MBPTA (blue).

We see that on probabilistic hardware, the normal RapiTime WCET value (pink) is exceptionally pessimistic because RapiTime’s normal analysis will construct a WCET that consists of all observed cache misses assuming that they have a high probability of occurring together. Whereas, the HyPTA line (green) is able to take into account the fact that cache misses have an independence and therefore the probability of them occurring at the same time is much lower.

Note that MBPTA applied on the test harness provides results (blue) that are more optimistic than the HyPTA results (green). This indicates that either MBPTA in this case does not guarantee complete path coverage or that HyPTA is more pessimistic than MBPTA. The two options are currently under study.
4.9 Software-based Randomisation

Differently to previous sections, we now evaluate the effect of performing software randomisation as described in D1.2 instead of randomising the execution time by hardware means only. In particular, we compare our single-level cache architecture implementing random placement and random replacement with no software randomisation (\textit{HW-RP + HW-RR} for short) with a single-level cache architecture implementing modulo placement and random replacement with both function and stack software randomisation (\textit{SW-RP + HW-RR} for short) where randomisation is performed only at the beginning (no re-randomisation is performed). For this experiment we use the PSB-SW synthetic benchmark.

\textbf{pWCET estimations for the synthetic benchmark}. First, we have tested whether execution times are i.i.d. by using the same tests described before. As expected, all measurements have passed the i.i.d. tests as shown in Table 4.17. We have also computed the minimum number of runs required for this experiment analogously to the hardware-only solution and obtained that the number of runs required is 350.

Figure 4.17 shows the pWCET estimations with MBPTA-EVT for both configurations, \textit{HW-RP + HW-RR} and \textit{SW-RP + HW-RR}. As shown, software randomisation produces higher pWCET values. The reason for such behaviour is twofold: On the one hand, software randomisation introduces some overhead due to the randomisation technique itself; on the other hand, software randomisation can produce fewer cache conflict layouts than hardware randomisation. This is so because it is applied at a coarser granularity, so fewer memory objects are randomised. As a consequence, the probability of bad cache conflict layouts is higher, thus increasing both the average execution time and the pWCET. Results for $10^{-13}$ and $10^{-16}$ exceedance probabilities are provided in Table 4.18. We can see that software
randomisation produces pWCET estimations in the order of 5× those of purely hardware randomisation.

In order to understand how significant is the overhead due to performing software randomisation and the impact of the number of layouts, we have repeated the same experiment on top of random-replacement fully-associative caches where the number of layouts cannot be increased beyond the number produced by the hardware itself. Therefore, in such an experiment all the execution time increase with respect to non-software randomisation is due to the software randomisation technique. Those results are shown in Figure 4.18. The overhead of software randomisation is around 1,000,000 cycles. The remaining execution time increase (around 500,000 cycles) is due to the lower number of cache conflict layouts obtained with software randomisation. Note that the relative overhead of software randomisation is directly related to the granularity at which it is applied: increasing the number of functions or decreasing the execution time increases the relative overhead. Conversely, increasing the number of functions, increasing the number of times they are called or increasing the execution time increases the number of cache conflict layouts, thus decreasing the pWCET. In summary, having more functions increases the overhead but also increases the quality of the software randomisation. In particular, in our example four functions called 100 times each provide enough randomisation so that re-randomisation is not required. Similarly, programs with high execution times make software randomisation overheads pay off and increase the number of cache conflict layouts, thus further decreasing the pWCET.
Figure 4.18: Overhead in the pWCET estimations with MBPTA-EVT for the synthetic benchmark for the software randomisation techniques
Acronyms and Abbreviations

- CDF: Cumulative distribution function.
- ECDF: Empirical cumulative distribution function.
- ETP: Execution time profile.
- EVT: Extreme value theory.
- GEV: Generalised extreme value distribution.
- HW: Hardware.
- ICDF: Inverse cumulative distribution function.
- IMA: Integrated modular avionics.
- ISA: Instruction set architecture.
- MBPTA: Measurement-based probabilistic timing analysis.
- PRNG: Pseudo-random number generator.
- PTA: Probabilistic timing analysis.
- pWCET: Probabilistic worst-case execution time.
- SPTA: Static probabilistic timing analysis.
- TLB: Translation look-aside buffer.
- UoC: Unit of composition.
- WCET: Worst-case execution time.
References


