A Confidence Assessment of WCET Estimates for Software Time Randomized Caches

Pedro Benedicte[‡,†], Leonidas Kosmidis[‡,†], Eduardo Quiñones[†], Jaume Abella[†], Francisco J. Cazorla[†,*]
[‡]Universitat Politècnica de Catalunya (UPC), Spain
[†]Barcelona Supercomputing Center (BSC-CNS), Spain
[*]Spanish National Research Council (IIIA-CSIC), Spain

***Abstract*—Obtaining Worst-Case Execution Time (WCET) estimates is a required step in real-time embedded systems during software verification. Measurement-Based Probabilistic Timing Analysis (MBPTA) aims at obtaining WCET estimates for industrial-size software running upon hardware platforms comprising high-performance features.**

MBPTA relies on the randomization of timing behavior (functional behavior is left unchanged) of hard-to-predict events like the location of objects in memory – and hence their associated cache behavior – that significantly impact software's WCET estimates. Software time-randomized caches (*sTRc*) have been recently proposed to enable MBPTA on top of Commercial off-the-shelf (COTS) caches (e.g. modulo placement). However, some random events may challenge MBPTA reliability on top of *sTRc*. In this paper, for *sTRc* and programs with homogeneously accessed addresses, we determine whether the number of observations taken at analysis, as part of the normal MBPTA application process, captures the cache events significantly impacting execution time and WCET. If this is not the case, our techniques provide the user with the number of extra runs to perform to guarantee that cache events are captured for a reliable application of MBPTA. Our techniques are evaluated with synthetic benchmarks and an avionics application.

I. Introduction

Critical real-time embedded systems increasingly use high-performance hardware features (e.g., cache memories) to timely run complex critical software. However, those hardware features challenge deriving Worst-Case Execution Time (WCET) [3] estimates, needed for timing validation and verification purposes. Measurement-Based Probabilistic Timing Analysis (MBPTA) [9] has been recently proposed as an industrial-friendly timing analysis method that allows obtaining reliable and tight WCET estimates on top of complex hardware. MBPTA, which has been positively assessed in avionics case studies [22], [23], derives a probabilistic WCET (pWCET) distribution function tightly upper-bounding the execution time of the program under analysis during operation. This pWCET value has to fulfill the software assurance standards specified for each different application domain.

MBPTA uses as input a collection of execution time observations – whose number is kept in the order of some hundreds – captured during the analysis phase [9]. Then, MBPTA applies Extreme Value Theory [11], [18] (EVT) to derive the pWCET distribution that holds during operation. The challenge lies in guaranteeing that the observations obtained at analysis time capture those events that can impact execution time during operation, and so pWCET estimates [8]. We refer to those events as *events of interest* (*eoi*). In this respect, EVT has to be understood as a technique to predict pathological combinations of those observed events in the analysis-time measurements. In general, EVT cannot predict the appearance of unobserved events since their impact on execution time can be arbitrarily large [1]. To cover this gap, MBPTA builds an argument on representativeness by means of i) either injecting randomization in the timing behavior of certain hardware resources (e.g. caches and buses) so that it is possible to determine the probability of their worst behavior to be captured in the analysis-time measurement runs; or ii) making resources to work on their worst latency so the analysis time measurements capture the worst timing behavior that those resources may have during operation.

For caches, one of the resources with highest impact on pWCET, several customized hardware time-randomized caches (hTRc) have been shown to comply with MBPTA requirements [13], [22]. Since those solutions have not been implemented in commercial processors yet, in this paper we focus on the so called software time-randomized caches (*sTRc*) [14]. *sTRc* build on top of (common) deterministic set associative caches – with W ways and S sets – for instance deploying modulo placement and LRU replacement. A user-level library makes program data and code to be randomly allocated in memory across runs. This results in data and code being mapped in random cache sets across runs. Ultimately, this causes data and instruction conflicts to have a random nature, making deterministic caches to have the probabilistic properties required by MBPTA [14].

In the processor architectures studied so far which resemble that of the LEON3 [21], cache placement has been regarded as the only hardware block threatening MBPTA's reliability [1]. An *eoi* can occur when more than W program objects (i.e. data or code) are placed in the same set [1]. In that case, those objects do not fit into the cache set, which may lead to an abrupt increase of the number of misses w.r.t. the case where the number of objects is up to W [1], [19]. In this scenario it is mandatory to observe the cache *eoi* at least once in the measurements collected during the analysis phase with sufficient confidence (defined for each domain in its safety standards) to preserve MBPTA's reliability. Failing to do so would go against principles for a correct MBPTA application, namely *capturing with analysis-time measurements those events with high potential impact on execution time*.

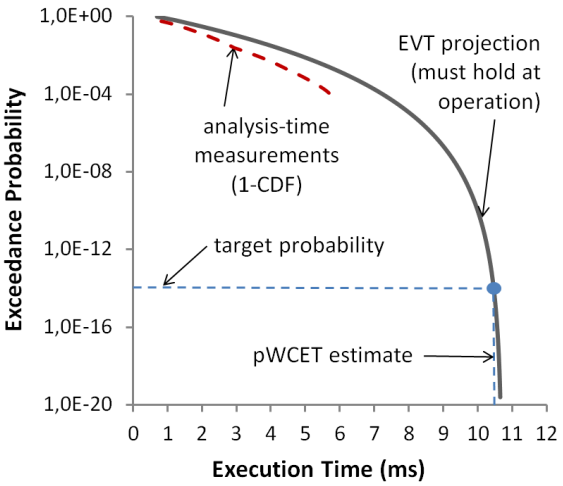

Fig. 1. Synthetic program's CCDF (log scale) [1].

This paper proposes an approach to compute the probability of the cache *eoi* for set-associative *sTRc*, called P_{eoi}. This is a fundamental step to gain confidence on pWCET estimates obtained with MBPTA. Given the set of objects to be allocated in memory and their size – which are known at design time – our techniques determine whether with the number of runs (R) carried out at analysis the cache *eoi* will be captured with a sufficiently high probability. Otherwise, we determine the number of runs required at analysis time to ensure that the relevant *eoi* is captured. Our approach focuses on programs that have homogeneously accessed objects so the impact on timing of any $W+1$ objects mapped to the same set is similar. For instance, this is the case for the instruction access of several loop-based control applications.

We provide an evaluation based on synthetic benchmarks for sensitivity analysis and a real avionics application [22], which confirm that our approach tightly estimates P_{eoi}.

As an example let us assume 3 functions, each occupying 1 cache line. Further let us assume that from run to run those functions are randomly mapped to memory which translates into them being assigned to a random set in cache. In an 8-set 2-way set-associative cache, the probability of those 3 functions to overlap in the same set (P_{eoi}), hence causing an increase in execution time, is $P_{eoi} \simeq (1/8)^3$. The probability of capturing this on $R = 1,000$ is too low ($P_{obs} = 85.84\%$). This can affect the confidence of MBPTA, since there is no guarantee that the events of interest will happen in the analysis runs. The proposed solution, explained in Section III is to increase the number of runs to $R' = R + 9,600$ so that the probability of not observing this event of interest is low enough (e.g., $< 10^{-9}$).

The rest of this paper is structured as follows. Section II explains the addressed problem and introduces the basic concepts. Section III describes the techniques used to compute P_{eoi} and how to obtain R'. Section IV presents the experimental results based on synthetic sequences and a real avionics case study. Section V describes some related work. Section VI exposes our main conclusions.

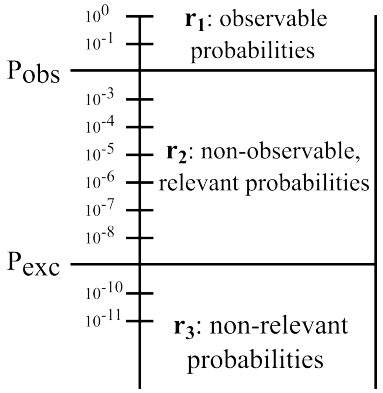

Fig. 2. MBPTA probabilities of interest.

II. Problem Statement

Next we introduce the notion of MBPTA representativeness and how it is affected by the use of time randomized caches.

A. MBPTA representativeness requirements

MBPTA delivers a distribution function upper-bounding the execution time (pWCET) of the program analyzed during operation. Such distribution function can be described in the form of a complementary cumulative distribution function (CCDF or 1-CDF), also known as tail distribution. An example is shown (Figure 1) for a program when collecting $R = 1,000$ execution time observations and applying EVT. With R observations one could estimate the pWCET for probabilities in the range $1/R$ at most. However, the pWCET curve delivered by EVT upper-bounds the probability of exceeding any execution time for arbitrarily low probabilities (e.g., 10^{-15} per run).

Execution time conditions when executing the program (e.g., initial instruction cache state) during operation are, generally, unknown or hard to produce at analysis time. Therefore, pWCET estimates provided by MBPTA are obtained under specific execution conditions intended to lead to equal or higher execution times than those that can occur during operation (e.g., empty instruction cache). Further, the correct application of EVT [8], [9] requires that execution time measurements collected at analysis time are independent and identically distributed, which can be obtained using a MBPTA-compliant platform [8]. Further, MBPTA requires that the execution time measurements used as input for EVT are *representative*, meaning that they include information about all *relevant events* affecting execution time. In particular, EVT is a powerful statistical tool able to predict how observed events can combine and predict the probabilities of those combinations. However, EVT cannot account for events that have not been observed in the execution times used for the prediction [1]. Relevant events of interest are those that can occur with a sufficiently low probability not to be captured in the analysis-time measurements, but with a sufficiently high probability not to be negligible. In the context of software randomization, the event of interest (*eoi*) is the random placement produced by software means [14]. MBPTA considers two probability thresholds as shown in Figure 2.

- The *exceedance probability* (P_{exc}). Events with lower probability to occur than P_{exc} during operation can be regarded as irrelevant. P_{exc} relates to the corresponding safety standard and the assurance/integrity level of the program under analysis, see Section II B.
- The *observable probability* (P_{obs}). For events with occurrence probability higher than P_{obs}, the probability of not observing them in the R runs collected at analysis time is negligible (e.g., $P_{coff} = 10^{-9}$). P_{obs} is obtained as $1-(1-P_{eoi})^R$, where P_{eoi} is the probability of the *eoi* and R the observations collected at analysis time. P_{coff}, as P_{exc}, relates to the corresponding safety standard and the assurance level of the program. For $R = 1,000$ and $P_{coff} = 10^{-9}$, the inequation $P_{coff} \geq (1-P_{eoi})^R$ holds when $P_{eoi} \geq 0.021$. Thus, $P_{obs} = 0.021$.

P_{exc} and P_{obs} define three probability ranges for P_{eoi}:

- *r1*: P_{eoi} is sufficiently high so that the *eoi* will be captured with R runs at analysis time. In other words, $P_{eoi} \geq P_{obs}$.
- *r2*: P_{eoi} is lower than P_{obs} so the *eoi* cannot be captured with R runs at analysis time, but can occur with sufficiently high probability during operation.
- *r3*: P_{eoi} is low enough so that the *eoi* may occur during operation only with negligible probability in relation to safety standards and the assurance level of the program.

In this paper, for *sTRc* we provide means to determine P_{eoi}. If it falls in $r1$ and $r3$ the application of MBPTA with R runs can be deemed as reliable. If it falls in $r2$, we derive how many runs are needed so that P_{eoi} moves to $r1$ and the application of MBPTA is, hence, reliable. Thus, our approach is key to keep reliability of MBPTA on top of *sTRc*.

B. Exceedance probability and safety standards

Safety standards define a probability of failure for different integrity levels, for instance in DO178B/C [20] (avionics) or ISO26262 [12] (automotive). This probability of failure is only applicable for hardware random faults, not for software. The software verification process consists in a qualitative process in which "enough" evidence must be collected about the software not failing during operation. In MBPTA, the pWCET estimates are defined with an exceedance threshold, which upper bounds the possibility of failure of a certain task. This exceedance threshold upper-bounds the residual risk in the verification process for software applications.

When using conventional measurement based techniques on regular caches (with modulo placement), the mapping of the objects in cache relies on the user for testing the different possible cache layouts, which will lead to different (possibly high) execution times. This risk is qualitatively assessed by the tests the user makes, which can be complex depending on the software being analyzed. With MBPTA and *sTRc*, the different cache layouts are randomly explored. This occurs because in every run program objects (data and code) are randomly mapped to memory and hence, assigned to random cache sets. Therefore, the analysis of the different possible mappings does not rely on the ability of the user to create different memory mappings, but on the confidence that pWCET gives with a certain number of analysis runs. In this paper we explain how to gain this confidence by increasing the number of analysis runs when needed. When using MBPTA the probability that the pWCET estimates are proven not to be exceeded can be set to an arbitrarily low number (e.g. 10^{-15} per program run).

The only difference MBPTA introduces in the certification process is the replacement of the user's ability to qualitatively assess the residual risk by the systematic and quantitative method to upper-bound the residual risk.

C. sTRc-related representativeness challenges

To the best of our knowledge representativeness issues for MBPTA have only been discussed in [10], [19], where authors show that MBPTA may produce unreliable pWCET estimates sporadically for hardware time-randomized caches (*hTRc*). The HoG approach [1] is presented to address this issue for *hTRc*. It is shown [1] that the *eoi* corresponds to the case where the number of addresses competing for the space in one cache set is higher than W, i.e. the cache associativity. This results in a *cache event of interest* with frequent evictions when addresses are accessed often and in an interleaved way. Note that, if the number of addresses does not exceed W, they will end up fitting in a cache set[1].

We assume that the execution time impact of mapping any $K > W + 1$ addresses to the same set is similar. This happens often for the instruction addresses, since programs are usually accessed homogeneously inside a main control loop. Our approach could also be applied to those objects (or elements of them) that are identified to cause higher impact on execution time if mapped to the same set. As part of our future work we plan to devise a method to identify those objects/elements with higher execution time impact.

In summary, a reliable application of MBPTA requires proving that either such cache event of interest (more than W addresses competing for the same set) occurs with negligible probability during operation, or with sufficiently high probability at analysis time to account for it. For *sTRc* no previous study has been done on this matter and this paper covers this gap. Note that, unlike for *hTRc* [1], [7], for *sTRc* the probability of a given object to be mapped to a set depends on the sets to which previous objects where mapped, which in turn affects the computation of P_{eoi}. We analyze in detail this dependence and propose two methods, one theoretical and one empirical, to derive P_{eoi} for *sTRc*.

III. Models to Derive P_{eoi}

sTRc [14] are implemented by deploying software randomization on top of conventional caches, e.g. modulo placement and LRU replacement. Software randomization is a software mapping layer that allocates objects in memory in random locations across runs.

[1]For the sake of this explanation, we assume an addressable unit size matching cache line size. In reality usually the addressable unit is smaller. In that case, cache *eoi* occurs when the number of lines accessed mapped to a set exceeds its capacity.

TABLE I
BASIC NOTATION

$\mathcal{O}$	Sequence of objects to allocate
R	Number of runs carried out by MBPTA at analysis
S, W	Number of sets and ways (respectively) in cache
cl_b	Size in bytes of a cache line
m_l, m_b	Memory size in cache lines and bytes respectively
set_l	Number of memory lines mapped to a set

In order to explain the models to derive P_{eoi}, we use the notation defined in Table I. With *sTRc*, in an 'infinite-size' memory, when a memory object is randomly allocated in memory, the address assigned to it, $@_x$, is also random. Hence, the set where $@_x$ is placed is also random since the function $@_x \mod S$ leads to a random number between 0 and $S-1$ when $@_x$ is random. As a result, in an infinite-size memory the probability of an object to be assigned to a given set, i.e. $Pset$ is given by Equation 1, which matches that for *hTRc* [1]:

$$Pset = \frac{set_l}{m_l} = \frac{1}{S} \tag{1}$$

In reality, with finite-size memory (with m_b bytes) *Pset* computation varies. In terms of cache lines the memory can allocate up to $m_l = m_b/cl_b$ lines. The number of lines mapped to each set is $set_l = m_l/S$. Note that lines mapped to a given cache set are not consecutive in memory with modulo placement, instead consecutive cache lines in memory are mapped into different (consecutive) sets in cache.

Interestingly, the memory objects of a program are allocated sequentially and, obviously, the space allocated to an object cannot be allocated to others. This creates a dependence among the $Pset$ of each object. Assuming single-line objects, the first allocated object has the same probability to be mapped to any set s_i as given by Equation 1. After the allocation of the first object, the number of lines that can be allocated to s_i, i.e. l_i, is reduced by one: $l_i = set_l - 1$. This decreases the probability of a second object to go to the same set as follows $Pset_i^{1,1} = (set_l - 1)/(m_l - 1)$.

$Pset_i^{x,y}$ is the probability of allocating an object in s_i given that y objects have already been allocated, from which x have been allocated to s_i. The probability of allocating the second object to any other set $Pset_j : j \neq i$ is $Pset_i^{0,1} = set_l/(m_l - 1)$. In general, after k objects have been allocated, the probability of a new object to be allocated to a set in which l objects have been allocated, with $l \leq k$, is given by Equation 2. Note that if $l \geq n_{ls}$ then $Pset = 0$.

$$Pset^{k,l} = \frac{set_l - l}{m_l - k} \tag{2}$$

Hence, there is a dependence between the specific sets where previous objects have been allocated and the probability that the current object is assigned to a given set. Notably, this dependence only occurs for *sTRc*, while for *hTRc* each object has a probability $1/S$ to be mapped in a given set [7].

A. Definitions and notation used to derive P_{eoi}

In order to derive P_{eoi} we focus first on single-line objects, i.e objects occupying one cache line, and later we focus on multi-line objects which comprise consecutive elements occupying one cache line each element.

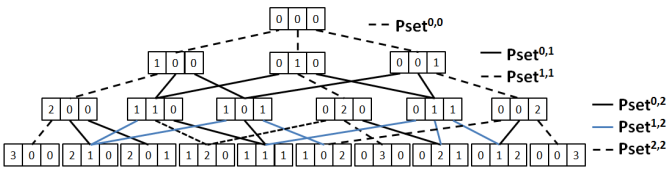

Fig. 3. Probability tree with *sTRc* when allocating 3 single-line objects into a 3-set cache. Leaves nodes are the allocation scenarios with cardinality 3.

Interestingly, the use of dynamic memory allocation is in general not allowed in critical real-time systems. Hence, *at analysis time the size of all program objects (e.g. functions, stack frames) are known*. We refer to the list of objects to allocate as $\mathcal{O}$. Note that allocation order is also known at analysis time. For instance, functions are usually allocated in the order they are found in the binary.

Definition (Allocation Scenario)**.** *An allocation scenario stands for a specific mapping of single-line objects (or their elements if multi-line) to cache sets.*

Definition (Cache event of interest)**.** *Cache events of interest correspond to those allocation scenarios where at least one cache set contains a number of single-line objects (or elements for multi-line objects) higher than W.*

For a program with 3 single-line objects ($|\mathcal{O}| = 3$) and a 2-way 3-set cache (W=2, S=3), the possible allocation scenarios are $\mathcal{A} = \{(0,0,3), (0,1,2), (0,2,1), (0,3,0), (1,0,2), (1,1,1), (1,2,0), (2,0,1), (2,1,0), (3,0,0)\}$, from which the cache events of interest are $(3,0,0), (0,3,0)$ and $(0,0,3)$.

Definition (Probability of the event of interest)**.** *The probability of the event of interest (P_{eoi}) is obtained by adding the probabilities of all cache events of interest.*

B. Theoretical (intractable) model of P_{eoi}

The different scenarios that can occur and their associated probabilities can be derived analytically by expanding the allocation tree in Figure 3 for each new object allocation. P_{eoi} can be obtained as the addition of the probabilities of those allocation scenarios where there is at least one cache set with a number of elements e so that $e > W$. However, this process grows exponentially and leads to a number of leaves in the order of $S^{\mathcal{O}}$, which is computationally intractable. For instance, for a 64-set cache and a 50-object sequence, the number of allocation scenarios (with repetitions) that could be reached would be around 10^{90} (note that the estimated number of atoms in the Universe is around 10^{82}). This makes this model infeasible to be implemented in reality and calls for an alternative model.

With *sTRc* **multi-line memory objects** are randomized atomically, that is, the first element of the object is assigned a random location in memory while the rest of the elements occupy consecutive memory positions. When modulo is used, multi-line objects are allocated into consecutive sets in cache. Hence, the probability of going from an allocation scenario to another is given by the probability that the first element of the

TABLE II
CONFIDENCE INTERVAL WIDTH.

Confidence	10,000 runs	1M runs	100M runs
0.99	0.013	0.0013	0.00013
$1-10^{-9}$	0.030	0.0030	0.00030

object goes to a particular line, since the placement of all the other elements of the object is determined by the placement of the first element.

C. Empirical model of P_{eoi}

The main approach we follow to derive P_{eoi} is based on Monte-Carlo simulations using an algorithm that processes the object sequence several times allocating each object in a random location in each simulation. Each run in which objects are randomly allocated to memory addresses (and hence sets) represents a Monte-Carlo simulation. This allows deriving the probability of each allocation scenario and hence P_{eoi}, for both single and multi-line objects.

While the Monte-Carlo method does not provide an exact result, we can statistically derive the confidence and accuracy of the result depending on the number of simulations performed. In particular, for each estimate obtained we compute the confidence interval ($value \pm variation$) assuming a certain degree of confidence. Determining this narrow confidence interval allows us making worst case assumptions, with little impact. That is, if any value in the range determined by the confidence interval is in $r2$ we assume that the value is not safe, and we compute the number of extra runs needed to guarantee that all values in the confidence interval fall in $r1$ with a sufficiently high probability.

As an example, for an arbitrary cache event whose actual probability is $P_{eoi} = 0.5172$, Table II shows the width of the confidence interval for different confidence values (0.99 and $1-10^{-9}$) as we increase the number of Monte-Carlo simulations. We observe that for $1-10^{-9}$ with $10,000$ runs is 0.030, which is $\pm 0.058\%$.

Based on this analysis, we perform 100 million simulations that require less than 15 minutes in a regular laptop and provide narrow confidence intervals.

In Figure 4 we can see the P_{eoi} range given by the Monte-Carlo simulations. For this example the confidence level is set to $1-10^{-9}$ and we have made 100 million simulations. Since the P_{eoi} range is really small (less than 2% variation at any point), in Figure 4 we can see the three lines defining the interval overlapping graphically.

D. Using P_{eoi}

Building on top of P_{eoi} and given i) a set of objects to allocate with their respective size, $\mathcal{O} = (o_1, o_2, ..., o_n)$, and ii) the number of execution time measurements collected at analysis time R; we can assess whether P_{eoi} falls in probability range *r2* (see Figure 2). In that case, we can easily determine the increased number of runs R' required to ensure the *eoi* is observed with sufficiently high probability.

The number of runs (R') required to ensure with a given degree of confidence (in line with P_{coff}) that the *eoi*, whose probability is P_{eoi}, will be observed is obtained with the following equation: $(1-P_{eoi})^{R'} \leq P_{coff}$. From this equation we can derive R' as follows:

$$R' \geq \frac{log(P_{coff})}{log(1-P_{eoi})} \quad (3)$$

Therefore R' is the minimum number or runs needed to ensure that the cache *eoi* are probabilistically captured in the analysis measurements. This is a fundamental step to ensure that MBPTA-derived pWCET estimates are reliable.

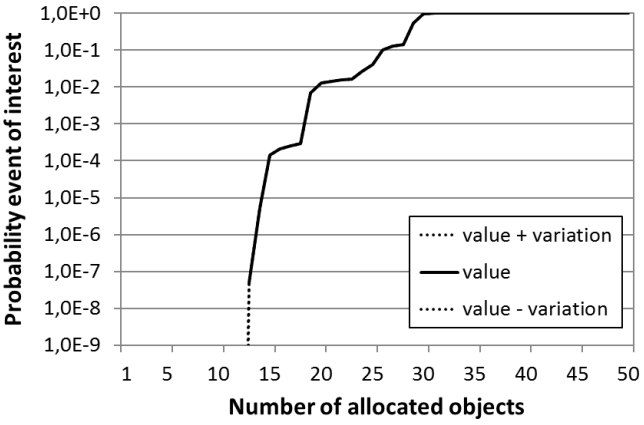

Fig. 4. P_{eoi} confidence interval for the Monte-Carlo simulations.

IV. EXPERIMENTAL RESULTS

We evaluate our empirical model to obtain P_{eoi} and we compare it against the theoretical model. For that purpose we use i) synthetic object sequences and ii) object sequences coming from a real avionics case study [22].

In our experiments we use two cache sizes, a small one (*scache*) and a big one (*bcache*). *scache* has been chosen to be particularly small to deploy the theoretical model for comparison purposes. As the cache size and/or the number of objects increases, the theoretical model becomes intractable as explained before. The *scache* is a 2KB 4-way 64B/line cache (so with 8 sets), whereas the *bcache* size is 128KB 8-way 64B/line cache (so with 256 sets). These values are representative of first and second-level caches (when partitioned). For instance, caches of the Cobham Gaisler NGMP [4] or the ARM Cortex A7 [6] are within this range. We have used a 20MB memory size for our experiments. Our results evidence negligible P_{eoi} variance for different memory sizes, so we do not report results for different memory sizes.

We consider two different types of sequences of objects to allocate: synthetically generated and based on a real avionics case study. For the former we consider two types of objects regarding their size: small objects (between $32B$ and $512B$) and big objects (between $1KB$ and $32KB$). By mixing objects of these two types we generate three types of object sequences. The *smallSeq* comprises 75% of small objects and 25% of big objects. The *bigSeq* comprises 25% of small objects and 75%

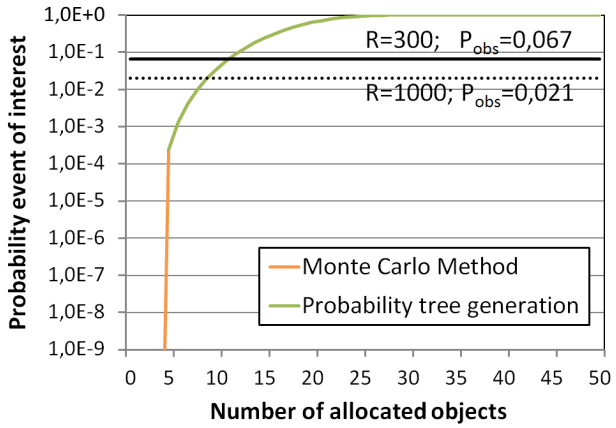

Fig. 5. Comparison of the 2 methods

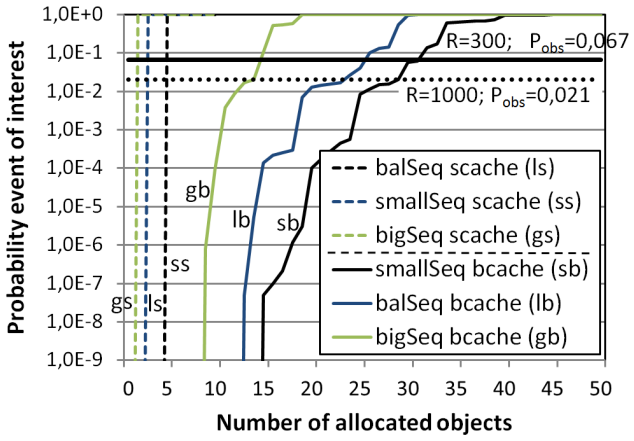

Fig. 6. P_{eoi} for different cache and sequence sizes for benchmarks.

big objects. And the *balSeq* balances small and big objects. All sequences have a size varying from $|\mathcal{O}| = 10$ to $|\mathcal{O}| = 600$.

A. Synthetic Sequences

In this experiment we focus on a *scache* setup and single-line objects. Figure 5 shows how the probability of the cache *eoi* varies as more objects are allocated. As a reference technique we implement the probability tree (as shown in Figure 3). In Figure 5 we represent with horizontal lines P_{obs} for two different numbers of runs: $R = 300$ and $R = 1,000$. The former is the lowest number of runs required by the method [9] and the latter is a typical value used for many programs [9]. Despite the probability tree provides exact results, its memory and execution time requirements prevent using it in the general case. We rather use it in this small example to show the accuracy of the Monte-Carlo method.

We observe no noticeable difference between both methods, thus providing evidence of the accuracy of our analytical and empirical methods. As expected, the higher the number of allocated objects the higher the probability of the *eoi*. Interestingly for sequences smaller than 5 objects the probability of the cache *eoi* is below 10^{-9} that we use as the reference exceedance probability, P_{exc}. Meanwhile, when 12 objects are allocated the probability of the *eoi* is above P_{exc}.

In a second experiment, shown in Figure 6, we analyze the impact of cache size (*scache* and *bcache*) and the sequence size (small, bal and big). For the *scache* setup the probability of the *eoi* reaches P_{obs} for both values of R (300 and 1,000) with less than 5 allocated objects in all 3 object sequence types. This occurs because the *scache* is small compared to the object sizes used, and with few objects the entire cache is filled. It can also be observed that, as expected, the bigger the objects are, the fewer objects are needed to reach P_{obs}. In fact, the line for the *bigSeq* raises so fast that cannot almost be observed in Figure 6. For the *bcache* setup, instead, we observe that more objects are needed in order to reach P_{obs}. Anyway, as soon as 30-35 objects are allocated MBPTA provides reliable results with its default number of runs.

As an example of the application of the results in Section III-D, let assume the *bcache* setup, $R = 1,000$ as the number of runs under MBPTA and an object sequence of $|\mathcal{O}| = 25$. For this scenario, we can see in Table III the P_{eoi} and number of new runs R' needed for each cache configuration.

TABLE III
NUMBER OF EXTRA RUNS R' DEPENDING ON P_{eoi}.

	smallSeq	***balSeq***	***bigSeq***
P_{eoi}	0.0085	0.0407	1.0000
R'	2,428	499	1

B. Avionics case study

We applied our technique to an industrial-size avionics application [22] consisting of around 5,000 functions varying from few bytes to 300KB each. The total size of those functions is 4.7MB if they are enforced to be aligned with cache line boundaries. In this experiment the focus is on code randomization, i.e. on the instruction cache. With instruction *sTRc* a software layer allocates functions code in memory in random positions across runs.

Characteristics. Many avionics systems build upon the Integrated Modular Avionics (IMA) concept [5], which defines how different subsystems can be integrated onto the same hardware platform, so that size, weight and power costs can be reduced. In the context of IMA, the system architecture implements temporal and spatial partitioning to avoid undesired functional and temporal interferences across different applications – especially in the context of mixed-criticalities. Temporal partitioning is generally implemented by partitioning time into scheduling units and using a static schedule generated offline. The **MA**jor **F**rame (MAF) is the hyper-period of all partitions. Each MAF is divided into a number of **MI**nor **F**rames (MIF) whose duration and period is identical. Time partitions are scheduled inside those MIF. Those time partitions contain processes, and each process contains a number of functions, that are the smallest unit of analysis possible. This is illustrated in Figure 7.

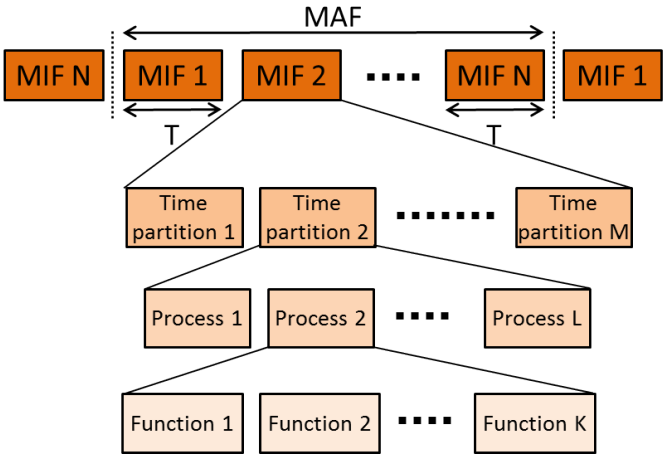

Fig. 7. Scheduling of functions, processes, time partitions and MIF within a MAF.

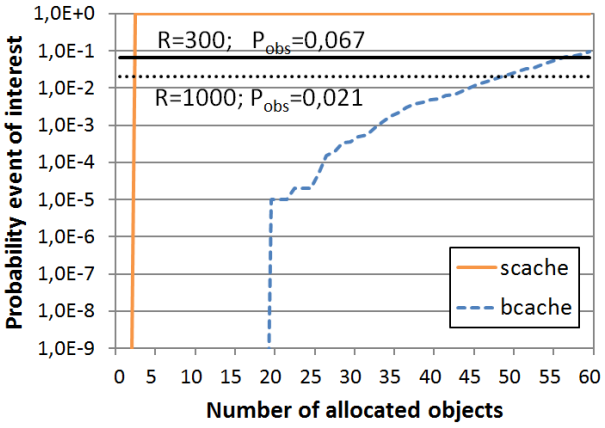

Fig. 8. P_{eoi} for different cache sizes for the avionics application.

We analyze two different applicability approaches of our model to the avionics application. When software randomization (and so pWCET estimation) is applied (1) at MAF (full application) granularity and (2) at finer granularity i.e. at MIF, partition, process or function level.

Full application granularity. In this case the avionics case study is analyzed with its 5,000 functions. Figure 8 shows that for the *scache* setup with as low as 2 objects allocated, the probability of the *eoi* is above P_{obs}. For the *bcache* we observe that between 45 and 60 object allocations are needed to reach P_{obs}. Hence, for this particular case example, the number of runs carried out $R = 1,000$ with MBPTA standard process were enough to ensure representativeness of the instruction cache events of interest. Notably this real case requires more object allocations than the synthetic object mixes because the average function size is smaller than our small objects. However, even with objects this small, the number of objects that must be allocated so that the event of interest is shown in the analysis runs is largely below the total number of objects in the application.

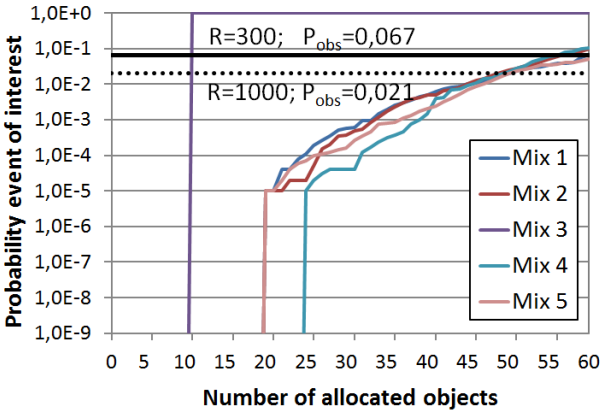

Fig. 9. P_{eoi} for the avionics case study for different (random) object allocations.

Analysis at fine granularity. Finer granularity than the full MAF may be used by end users to analyze the timing of their applications. However, there is not a strict rule on what the right granularity is. The finer the granularity is, the higher the control exercised on the execution paths traversed, on the execution time cost of each function or process, etc. On the other hand, finer granularity also implies adding more instrumentation and exercising a stronger control on the process to collect execution time measurements. Thus, there is a tradeoff between the amount of information that can be retrieved and the cost to obtain it.

These experiments evaluate the impact of the analysis granularity on P_{eoi} to allow the users find the most convenient granularity from a software randomization point of view. For that purpose we incrementally pick functions from the case study (randomly) and compute P_{eoi} and the number of runs needed to apply MBPTA reliably (R').

First, for the sake of illustration we have sorted the 5,000 functions randomly 100 times. Then, we have obtained P_{eoi} as objects are allocated. The value of P_{eoi} for the first 5 sortings is shown in Figure 9. In general, although some differences exist depending on the objects that form the unit of analysis, we observe that all sortings reach $r1$ (so $P_{eoi} \geq P_{obs}$) with around 50 objects.

TABLE IV
IMPACT OF THE NUMBER OF OBJECTS ON P_{eoi}.

Functions	10	20	30	40	50	60
P_{eoi}	0.0000	0.0100	0.0105	0.0165	0.0549	0.1606
Range	$r3$	$r2$	$r2$	$r2$	$r2$	$r1$
R'	-	2,061	1,959	1,243	367	-

In Table IV we summarize the average value of P_{eoi} across all 100 sortings, the range (i.e. r_1, r_2 or r_3) where that probability belongs to, and the number of runs R' that would be needed for different object counts if P_{eoi} falls in $r2$. If we assume that, for instance, $P_{exc} = 10^{-9}$ and $R = 300$, P_{obs} will be around 0.067. As shown in the table, P_{eoi} is virtually 0 with 10 objects, meaning that the probability of the *eoi* is so low that is not relevant ($r3$). Then, from $R = 20$ to $R = 50$,

P_{eoi} falls in $r2$, thus meaning that the number of runs R' needs to be higher than 300. In this example R' must be between $2,061$ (20 objects) and 367 (50 objects). Finally, P_{eoi} falls in $r1$ with 60 objects or more, so 300 runs suffice to guarantee a reliable application of MBPTA.

We have shown how our technique, building on the set of objects to be allocated in memory – usually known at design time – is able to determine whether more runs that the default R need to be performed to ensure than cache events of interest are captured.

V. Related Work

MBPTA has received significant attention in recent years. Academic works have compared the average performance of MBPTA using *hTRc* and static timing analysis (STA) using time-deterministic caches, with only 12% worse average performance for *hTRc* [13]. In terms of WCET other studies [2] show that MBPTA provides competitive WCET estimates when compared against STA techniques on top of time deterministic caches [16]. Further, since MBPTA is a measurement-based approach, it adapts faster to new processors [24]. Path coverage in the context of MBPTA has been addressed from a theoretical [17] and a practical [25] perspective for *hTRc*. How to extend these methods to *sTRc* and how to size the number of runs accordingly is part of our future work.

From a more industrial perspective, MBPTA's use in industry case studies (both avionics and automotive industries) has been positively assessed [15], [22], [23].

The problem of representativeness of the event of interest has been identified in some publications [1], [10], [19]. Although these representativeness matters are considered a risk by some authors [19], recent publications show how to mitigate the risk [1], [10]. Some techniques have dealt with this representativeness issue for *hTRc* [1], but this paper presents the first techniques to solve this problem in *sTRc*.

VI. Conclusions

The use of Extreme Value Theory in MBPTA is challenged by the fact that the execution time observations used for the prediction are those obtained at analysis time, while the predicted pWCET estimate must provide a reliable upper bound during operation. Evidence is required proving that execution time observations obtained at analysis time capture the impact of relevant events affecting execution time and that can arise during operation. Given the objects to be allocated for an application, we propose a method to compute the probability of cache events of interest for software time-randomized caches. Whenever that probability is high enough to be relevant and low enough so that high confidence on observing the event cannot be achieved, our method computes the extra number of runs needed to regain confidence on pWCET estimates provided by MBPTA. Given MBPTA's incremental assessment in industrial domains, our technique is a fundamental step to maintain confidence in the pWCET estimates obtained with MBPTA.

Acknowledgments

The research leading to these results has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under the PROXIMA Project (www.proxima-project.eu), grant agreement no 611085. This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2015-65316, the HiPEAC Network of Excellence, and COST Action IC1202: Timing Analysis On Code-Level (TACLe). Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

[1] J. Abella et al. Heart of Gold: Making the improbable happen to extend coverage in probabilistic timing analysis. In *ECRTS*, 2014.
[2] J. Abella et al. On the comparison of deterministic and probabilistic WCET estimation techniques. In *ECRTS*, 2014.
[3] J. Abella et al. WCET analysis methods: Pitfalls and challenges on their trustworthiness. In *SIES*, 2015.
[4] Aeroflex Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*, 2011.
[5] ARINC. *Specification 651: Design Guide for Integrated Modular Avionics*. Aeronautical Radio, Inc, 1997.
[6] ARM. ARM Cortex-A7 Processor Specification. http://www.arm.com/products/processors/cortex-a/cortex-a7.php.
[7] P. Benedicte et al. Modelling the Confidence of Timing Analysis for Time Randomised Caches. In *11th IEEE International Symposium on Industrial Embedded Systems*. IEEE, 2016.
[8] F.J. Cazorla et al. Upper-bounding program execution time with extreme value theory. In *WCET Workshop*, 2013.
[9] L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.
[10] Enrico Mezzetti et al. Randomized caches can be pretty useful to hard real-time systems. *LITES*, 2(1), 2015.
[11] W. Feller. *An introduction to Probability Theory and Its Applications*. 1996.
[12] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
[13] L. Kosmidis et al. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.
[14] L. Kosmidis et al. Probabilistic timing analysis on conventional cache designs. In *DATE*, 2013.
[15] L. Kosmidis et al. Containing timing-related certification cost in automotive systems deploying complex hardware. In *DAC*, 2014.
[16] L. Kosmidis et al. Probabilistic timing analysis and its impact on processor architecture. In *DSD*, 2014.
[17] L. Kosmidis et al. PUB: Path upper-bounding for measurement-based probabilistic timing analysis. In *ECRTS*, 2014.
[18] S. Kotz et al. *Extreme value distributions: theory and applications*. World Scientific, 2000.
[19] J. Reineke. Randomized caches considered harmful in hard real-time systems. *LITES*, 1(1), 2014.
[20] RTCA and EUROCAE. *DO-178B / ED-12B, Software Considerations in Airborne Systems and Equipment Certification*, 1992.
[21] http://www.gaisler.com/cms/index.php?option=com_content&task=view&id=13&Itemid=53. *Leon3 Processor*. Areroflex Gaisler.
[22] F. Wartel et al. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.
[23] F. Wartel et al. Timing analysis of an avionics case study on complex hardware/software platforms. In *DATE*, 2015.
[24] R. Wilhelm et al. The worst-case execution time problem: overview of methods and survey of tools. *ACM TECS*, 7(3):1–53, 2008.
[25] M. Ziccardi, E. Mezzetti, T. Vardanega, J. Abella, and F. J. Cazorla. EPC: Extended Path Coverage for measurement-based probabilistic timing analysis. In *RTSS*, 2015.