Measurement-Based Cache Representativeness
on Multipath Programs

Suzana Milutinovic
Universitat Politècnica
de Catalunya and BSC

Jaume Abella, Enrico Mezzetti
Barcelona Supercomputing
Center (BSC)

Francisco J. Cazorla
BSC and IIIA-CSIC

ABSTRACT
Autonomous vehicles in embedded real-time systems increase critical-software size and complexity whose performance needs are covered with high-performance hardware features like caches, which however hampers obtaining WCET estimates that hold valid for all program execution paths. This requires assessing that all cache layouts have been properly factored in the WCET process. For measurement-based timing analysis, the most common analysis method, we provide a solution to achieve cache representativeness and full path coverage: we create a modified program for analysis purposes where cache impact is upper-bounded across any path, and derive the minimum number of runs required to capture in the test campaign cache layouts resulting in high execution times.

CCS CONCEPTS
•Computer systems organization → Real-time systems;

KEYWORDS
Cache, randomization, multipath programs, probabilistic analysis

1 INTRODUCTION
Most new services in real-time products (e.g. cars) are totally or partially provided by software, making it one of the most important factors to increase competitive edge [11]. The increasing complexity of software, to manage huge and diverse types of data and for decision making, heavily complicates timing validation and verification. This is compounded by the use of performance-improving hardware features (e.g. caches) to cover unprecedented critical software’s performance needs, e.g. expected to increase by 100x in the next years in the automotive domain [34]. Increased hardware and software complexity hampers the confidence that can be put on WCET estimates derived by measurement-based timing analysis (MBTA), the most used timing analysis technique in critical real-time embedded systems [37]: the complex control-flow graph of software seriously complicates providing input data (vectors) that exercise all program paths; whereas caches create huge execution time variability since the way in which applications’ objects are mapped in memory – on which users have decreasing control – determines their cache layout and application’s performance.

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. Copyrights for components of this work owned by others than ACM must be honored. Abstracting with credit is permitted. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee. Request permissions from permissions@acm.org.

DAC ’18, San Francisco, CA, USA
© 2018 ACM. 978-1-4503-5700-5/18/06...$15.00
DOI 10.1145/3195970.3196075

The increased variability in the execution time of a task, whose distribution can have arbitrary variance and shape, has motivated several research works to use statistical techniques to derive bounds to execution time distributions. This has resulted in a variant of MBTA referred to as Measurement-based probabilistic timing analysis (MBPTA) [2]. MBPTA delivers a probabilistic WCET (pWCET) function (see Figure 1(a)) upper-bounding the execution time distribution (pETd) of the program at any exceeding probability. For instance, in Figure 1(a) the probability that execution time exceeds 25 ms is at most $10^{-10}$. MBPTA has been applied to avionics [35] and space [9] case studies and its impact on certification has also been addressed [33]. MBPTA has been complemented with solutions that inject randomization in program’s timing behavior to relieve the user from controlling those jittery resources increasing program’s execution time variability. This is achieved by hardware techniques (e.g. random arbitration policies and random placement/replacement techniques) that are now part of a commercial product for the space domain [5]; and software techniques that work at the compiler/linker level or source code level [15].

Randomization makes that all potential behaviors that a given jittery resource (e.g. caches) can have are naturally (and randomly) explored in every new test, enabling the derivation of probabilistic guarantees. As a result, the user is relieved from controlling memory mapping, which is arguably complex (in fact infeasible as hardware resources) can have are naturally (and randomly) explored in every new test, enabling the derivation of probabilistic guarantees. As a result, the user is relieved from controlling memory mapping, which is arguably complex (in fact infeasible as software complexity grows) [20]. As a side effect, the execution of a program on a randomized platform results in a probabilistic execution time distribution (pETd), see dashed line in Figure 1(a).

Time-aware Address Conflict (TAC) [24] works on top of MBPTA to factor in cache layouts building on randomization properties. TAC guarantees that the number of execution time measurements fed to MBPTA is large enough to observe the effect of all relevant cache placements, whose probability of occurrence may be sufficiently low not to be observed in the default number of runs of MBPTA [1]. If the number of runs satisfies this criteria, measurements are regarded as cache-layout representative. However, TAC has only been demonstrated on single-path analysis.

Path Upper-Bounding (PUB) builds an upper-bounding program [14] to automatically extend path coverage beyond the subset of paths exercised with user’s input vectors, effectively relieving the user from the enumeration of all paths. Nonetheless, the required minimum number of runs required for factoring in cache representativeness for a reliable application of MBPTA has not been explored in the scope of automatic full-path coverage, as provided by PUB.

Overall, no existing MBPTA solution attains cache representativeness and full path coverage simultaneously. We cover this gap by proposing the first approach to determine the minimum number of runs required in MBPTA-compliant platforms for MBPTA techniques delivering full path coverage automatically. In particular, this paper (1) devises a reliable combined application of PUB and TAC mechanisms by analyzing the relation between the memory...
address sequences in the original program and its modified (pubbed) version, thus identifying key properties that guide the joint application of PUB and TAC. This work also (2) provides evidence of the reliability of the approach, and (3) evaluates its cost on the Mälardalen benchmark suite.

2 BACKGROUND

MBPTA [2] applies Extreme Value Theory (EVT) [6, 16] on a set of execution time measurements, which must meet certain statistical properties (e.g. independence and identical distribution), and determines the best set of maxima values of the sample to be used to estimate the pWCET.

TAC. EVT can only estimate the combined effects of events observed in its input set (execution time values). For instance, some cache placements may produce arbitrarily higher execution times than others when the capacity of one or more cache sets is exceeded, and accesses to addresses mapping to those sets are interleaved with long reuse distances (e.g. in a round-robin fashion). EVT is unable to predict those events (and combinations thereof) when they are not observed in the execution time measurements passed as input [1]. TAC [24] analyzes the address sequence of the program under analysis, and determines the particular groups of addresses with larger impact on execution time when placed together in a set. Then, by correlating their impact with their estimated probability of occurrence, TAC determines the number of execution time measurements needed to guarantee that they are observed with a sufficiently high probability. For instance, the number of runs is sized so that the probability of missing such events is lower than $10^{-9}$, thus in line with the most stringent fault probabilities allowed for hardware components.

PUB builds a modified version of the program ($P_{\text{pub}}$) such that any path of $P_{\text{pub}}$ exhibits an execution time distribution that upper-bounds those of all paths in the original program $P_{\text{orig}}$.

$$V_{i,j} \in \text{paths}(P_{\text{orig}}) : F(P^j_{\text{orig}}(t)) \geq F(P^j_{\text{pub}}(t))$$

That is, the accumulated probability ($F(x)$) of any path $j$ of the original program for any execution time $t$ is equal or higher than for any path $j$ of the pubbed version of the program. Note that the extended program is used only to generate the pWCET at analysis, while at deployment time the original unmodified program is used.

PUB extends the source code of the original program with a compiler pass that infuses the code of each branch of conditional constructs (e.g. if-then-else, switch) with instructions and memory accesses so that each branch in the pubbed code includes equivalent cache access patterns in all of the branches of the original conditional construct\(^1\). By applying this concept recursively starting from the innermost conditional constructs, operations and memory patterns can be upper-bounded. Interestingly, PUB inserts the needed memory accesses with functionally-innocuous operations (e.g. loading data into a read-only register). PUB builds on input vectors triggering the highest loop bounds and all basic blocks (but not all paths), which is a common requirement of safety standards for code coverage. PUB relies on end users from having to control memory mapping, values operated and paths traversed, which are much less obvious from code inspection.

\(^1\)For clarity we refer to the sequence of addresses of one path, regardless of whether they are instructions or data since the reasoning provided in this paper applies to both.

![Figure 1: Basic concepts on MBPTA and PUB](image)

PUB relies on a key property of time-randomized caches that does not hold for time-deterministic caches: given a memory access sequence, adding any memory access in any position necessarily causes a worse probabilistic execution time distribution [14]. For instance, given an if-then-else where the ‘if’ branch contains the memory access sequence M\(_{if} = [\text{ABCA}]\) and the ‘else’ branch M\(_{else} = [\text{BACA}]\), the sequence M\(_{pub} = [\text{ABACA}]\) is an upper-bound for both (see Figure 1(b)). If we compare $M_{if}$ and $M_{pub}$, the first two accesses (A and B) will have the same probabilistic behavior. If the third access in $M_{pub}$ is a hit (A), cache contents are not altered and the rest of the sequence is identical to the remaining part of $M_{if}$. Hence, $M_{pub}$ increases execution time by 1 hit. Conversely, if the third access is a miss, the fifth access (A again) has higher probability to hit. Thus, $M_{pub}$ will experience 4 misses in the first 4 accesses and a likely-hit in the last access. Instead, $M_{if}$ will experience at most 4 misses, hence having a shorter execution time than $M_{pub}$. We refer the reader to the PUB technique for further proofs, details on instruction cache specifics and memory alignment issues [14]. Note that, this method is not compatible with time-deterministic caches [14] since, for instance, Least-Recently Used (LRU) replacement policies may lead to lower execution time by inserting accesses. E.g. in a 2-way cache the sequence (ABCA) experiences 4 misses (all accesses), whereas (ABACA) experiences only 3 (second and third occurrence of A are hits).

3 ACHIEVING FULL PATH COVERAGE AND CACHE REPRESENTATIVENESS

3.1 Difficulties integrating PUB and TAC

In order to reach full path coverage, PUB extended programs result in address sequences that highly likely differ from that of the original program. This complicates reasoning on TAC, i.e. on the number of runs to ensure that random cache layouts causing high execution times are captured. Intuitively, since PUB adds cache accesses to the original program, the number of runs required for TAC reduces, since the more the addresses in a program the more likely they conflict in the same cache set, and hence the lower the number of runs to randomly hit one of those cases. However, this is not the case. To illustrate so, let $M^j_{\text{orig}}$ be the address sequence of path $j$. Let ins($M, x$) be an operator to insert an address $x$ in a sequence $M$, which can be done in multiple ways, as long as the same ordering of address accesses in the original path is preserved. The sequence of addresses of a path $j$ in the pubbed program ($M^j_{\text{pub}}$) is the result of inserting several addresses, $n_j \geq 0$, in $M^j_{\text{orig}}$. 

---

\(V_{i,j} \in \text{paths}(P_{\text{orig}}) : F(P^j_{\text{orig}}(t)) \geq F(P^j_{\text{pub}}(t))\)

\(\forall i, j \in \text{paths}(P_{\text{orig}}) : F(P^j_{\text{orig}}(t)) \geq F(P^j_{\text{pub}}(t))\)
\( \forall j /\in /\text{paths}(P) : M_{j_{\text{pub}}}^1 = \text{ins}(\ldots(\text{ins}(M_{\text{orig}}^j, x_1), x_2), \ldots, x_n) \) (2)

No relationship can be established between the number of runs required to achieve representativeness in \( M_{\text{orig}}^j \) and \( M_{\text{pub}}^j \). That is, \( R_{TAC}(M_{\text{orig}}^j) \) can be higher/lower than \( R_{TAC}(M_{\text{pub}}^j) \), where \( R_{TAC}(M) \) is the minimum number of runs required for address sequence \( M \) as determined by TAC. This is shown with two examples.

3.1.1 \( R_{TAC}(M_{\text{orig}}^j) < R_{TAC}(M_{\text{pub}}^j) \). Let us assume a program with two paths, \( p_{\text{orig}}^1 \) and \( p_{\text{orig}}^2 \), with address sequences \( M_{\text{orig}}^1 = \{\text{ABCDEA}\}^{1000} \) and \( M_{\text{orig}}^2 = \{\text{ADEF}\}^{1000} \), and a cache with \( S = 8 \) cache sets and \( W = 4 \) ways. The exponent of the address sequences represents the number of times that those address sequences repeat. PUB application tries to minimize the number of addresses inserted to minimize the impact in the pWCET estimate. Hence, a potential result of applying PUB could be as follows:

\[
M_{\text{pub}}^1 = M_{\text{pub}}^2 = (\text{ins}(\text{ins}(M_{\text{orig}}^1, D), E))^{1000} = (\text{ins}(\text{ins}(M_{\text{orig}}^2, B), C))^{1000} = \{\text{ABCDEA}\}^{1000}
\] (3)

Since both \( M_{\text{orig}}^1 \) and \( M_{\text{orig}}^2 \) have 3 different addresses each, none of them can exceed the space in a cache set (4 ways), i.e., they cannot generate a cache layout resulting in high execution time. As a result, TAC does not impose a minimum number of runs higher than that required by the standard application of MBPTA. Note that, for instance, in the case of \( M_{\text{orig}}^1 \) addresses A, B and C will end up fitting in a cache set after, potentially, few random replacements [24].

Instead, \( M_{\text{pub}}^1 \) and \( M_{\text{pub}}^2 \) have 5 different addresses, so it exceeds the space in a cache set. For instance, if addresses A, B, C, D and E are randomly placed in the same set, they will experience at least 1000 misses due to the address not present in the cache set in each of the 1000 traversals of the address sequence. Hence, TAC determines that a sufficient number of runs is needed to guarantee that this sequence is not observed with a maximum configurable probability. The probability of observing such a sequence is exactly \( (1/S)^4 \leq 0.000244 \), so the probability of not observing it in \( R \) runs is \((1-0.000244)^R\). If such probability needs to be lower than \( 10^{-8} \) to be regarded as negligible w.r.t. the corresponding safety standard [24], then \( R \geq 84875 \). Hence, PUB address sequences may require a higher minimum number of runs than the original ones.

3.1.2 \( R_{TAC}(M_{\text{orig}}^j) > R_{TAC}(M_{\text{pub}}^j) \). Let us now consider that \( M_{\text{orig}}^1 = \{\text{ABCDEA}\}^{1000} \) and \( M_{\text{orig}}^2 = \{\text{ABCDEF}\}^{1000} \). In this case, and building upon the results obtained in the example before, each original path would require \( R > 84875 \) runs. \( M_{\text{pub}}^1 \) and \( M_{\text{pub}}^2 \) would be \( \{\text{ABCDEFA}\}^{1000} \), thus including 6 different addresses. Abrupt cache miss counts would require 5 out of the 6 addresses to be placed in the same set, whose probability is \((1/S)^4 \cdot 6 = 0.00146\). Therefore, \( R > 14138 \). Hence, PUB address sequences may require a lower minimum number of runs than the original sequences.

3.2 Sensible Application of PUB and TAC

Since no relation can be established between the minimum number of runs required by the original program and its pubbed version, our method relates both using only the pWCET and their execution time distributions. To that end we build on the following reasoning.

**Observation 1**: The probabilistic execution time distribution (pETd) obtained for any given path of the pubbed program is an upper-bound to the pETd of any path of the original program [14].

This observation is formalized in Equation 1. Note that the pETd refers to those that would be obtained by running the corresponding program paths an infinite number of times. Thus, while this observation would generally hold also for the statistical samples obtained from the pETd, it can only be guaranteed to hold for the reference distributions due to the statistical nature of samples.

**Observation 2**: The pWCET curve obtained from the sample of a pETd is (probabilistically) an upper-bound for the execution time distribution that has been sampled [2].

Under the appropriate setup for execution time collection, e.g. ensuring independence across experiments, and sources of randomization injected at hardware or software level, it has been shown that pWCET estimates upper-bound pETd [2, 27].

**Corollary 1**: From Observations 1 and 2, it follows that the pWCET curve obtained from a sample of any given path of a pubbed program upper-bounds the pETd of any path of the original program.

From Corollary 1, we make the following key observation:

**Observation 3**: Any path choice from the pubbed program is equally valid to derive a minimum number of runs and a reliable pWCET distribution that upper-bounds the pETd of all paths in the original program.

For instance, by applying TAC on an arbitrary path \( p_{\text{pub}}^j \) we obtain that the minimum number of runs needed for cache representativeness is \( R_{p_{\text{pub}}^j}^{\text{pub}} \). Since pWCET(\( p_{\text{pub}}^j \)) is, when it is derived from a sample of at least \( R_{p_{\text{pub}}^j}^{\text{pub}} \) runs, is an upper-bound for the pETd of any path of the original program, pWCET(\( p_{\text{pub}}^j \)) achieves both full path upper-bounding and cache representativeness.

Some observations must be made to complete the view on the approach taken to use PUB and TAC appropriately.

**Observation 4**: For two different paths \( j \) and \( k \) of a pubbed program, TAC results in \( R_{p_{\text{pub}}^j}^{\text{pub}} \) and \( R_{p_{\text{pub}}^k}^{\text{pub}} \) minimum number of runs, respectively. In general, no relation can be established between \( R_{p_{\text{pub}}^j}^{\text{pub}} \) and \( R_{p_{\text{pub}}^k}^{\text{pub}} \) since the insertion of the missing addresses in each path with \( \text{ins}(M, x) \) may not lead to identical address sequences.

For instance, if \( M_{\text{orig}}^1 = \{\text{ABA}\} \) and \( M_{\text{orig}}^2 = \{\text{ACA}\} \), by applying PUB we could obtain \( M_{\text{pub}}^1 = \{\text{ABC}\} \) and \( M_{\text{pub}}^2 = \{\text{ACB}\} \). In the general case, address sequences can be arbitrarily different, thus leading to different results when applying TAC. This is analogous to the scenario when relating the number of runs of the original and pubbed paths, described in Section 3.1.

Nevertheless, building on previous appreciation, even if a different pubbed path is used, the pWCET remains reliable and representative w.r.t. all paths in the original program. Building on the reasoning above, we also note that:

**Observation 5**: The pWCET estimates obtained for \( p_{\text{pub}}^j \) and \( p_{\text{pub}}^k \) cannot be related to each other. In other words, potentially, the pWCET estimate for path \( p_{\text{pub}}^j \) can be above or below that of \( p_{\text{pub}}^k \) for the full range of probabilities or only for a subset of them. In any case, this is irrelevant given that both pWCET estimates are reliable and representative upper-bounds of all paths of the original program.

**Corollary 2**: Since every single pWCET estimate for each pubbed path is a reliable and representative upper-bound for the pETd of all
paths of the original program, we can take as pWCET for any given exceedance threshold the lowest value across all pubbed paths.

Corollary 2 builds upon the fact that, the pWCET estimates of all paths are equally reliable and representative. Differences only relate to pessimism, so we can trade analysis cost for tightness by analyzing an increased number of paths of the pubbed program. While only one is strictly needed, the larger the number of paths analyzed, the higher the chances of obtaining tighter pWCET estimates. Note, however, that no guarantee exists on whether tighter pWCET estimates will be obtained since it could turn to be that the lowest one across the available paths is the first one we obtain.

We illustrate how our method works with an example based on we stick to its default input to set the loop bounds (15 integer elements), and explore the different type of input vectors (related to the degree of ordering of the elements to sort) that lead to the maximum number of iterations. In particular, 8 different cases lead to different paths triggering the maximum number of iterations.

### 3.3 A Practical Example

We illustrate how our method works with an example based on the binary search (bs) benchmark from the Mälardalen suite [10]. Details on the hardware setup are provided later in Section 4. Bs is a multipath program whose input determines the number of loop iterations and the actual path traversed. For the sake of illustration we stick to its default input to set the loop bounds (15 integer elements), and explore the different type of input vectors (related to the degree of ordering of the elements to sort) that lead to the maximum number of iterations. In particular, 8 different cases lead to different paths triggering the maximum number of iterations.

---

2 We omit details on PUB and TAC implementation due to space constraints and refer an interested reader to original works [14, 24].
variability is due to hardware effects, while for others execution time varies depending on input values. Applying PUB+TAC took approximately 15 minutes per benchmark on average.

4.1 Representative Number of Runs

First, we have collected the number of runs required for MBPTA convergence on the published version of the programs, \( R_{pub} \), and the number of runs on those same published versions as reported by TAC, \( R_{pub+tac} \). Results are shown in Table 2. As shown, despite in some cases the number of runs does not increase, TAC often requires \( R_{pub+tac} > R_{pub} \) to attain representativeness. In practice, \( R_{pub} \) runs often provide enough measurements from the tail of the execution time distribution, but not with enough confidence. Moreover, in some cases, as shown later, using \( R_{pub} \) runs instead of \( R_{pub+tac} \) may lead to pWCET estimates that do not account for some events that occur with significant probability. For completeness, we also report the number of runs required on the original version of the programs, applying neither TAC nor PUB, so only determined by MBPTA (\( R_{orig} \)). As shown, \( R_{pub+tac} > R_{orig} \), although there is no specific relation between \( R_{pub+tac} \) (or \( R_{pub} \)) and \( R_{orig} \), since they correspond to different programs in practice.

4.2 pWCET Estimates

For some benchmarks, the particular input data available leads to the worst-case path (or they are simply single path). Hence, for those benchmarks we can determine that any difference between the pWCET obtained with and without PUB is fully accountable to overestimation introduced to attain full path coverage. On the remaining benchmarks, instead, it is not possible to clearly tell apart how much of the pWCET increase corresponds to unobserved (worst) paths and how much is plain overestimation.

Some of the benchmarks evaluated are single path (the 6 rightmost ones in Figure 5): edn, insert sort, jfdc, matmult, fdct and ns. Others (the 4 leftmost ones in Figure 5) have multiple paths, but the default input vector used already triggers the worst-case path: bs, cnt, fir and janne. Finally, for crc we are unable to identify the worst-case path and, based on code inspection, we are highly confident that the default input does not trigger it.

Figure 5 shows the pWCET increase caused by the application of PUB and PUB+TAC w.r.t. the direct application of MBPTA with neither PUB nor TAC. First, we observe that the application of PUB increases pWCET estimates between 4% and 59% for the 4 leftmost benchmarks. In this case, since we know that we already exercised the worst-case path, we can tell this increase is pessimism due to PUB. For the 6 rightmost benchmarks PUB has no impact as they are single-path and hence, PUB application is innocuous. Finally, in the case of crc PUB leads to a 4.4x higher pWCET estimate (340% above that without PUB). While code inspection reveals that this increase captures unobserved paths, it is hard to tell apart how much of this increase is needed to attain path coverage and how much is pessimism incurred by PUB. However, this is the general situation we can expect in real applications: the worst-case path cannot be determined a priori and PUB accounts for it automatically.

When applying TAC on top of PUB, we observe a variety of different behaviors: (1) in most cases (e.g. bs and fir) PUB+WTC variation is relatively small. This occurs because \( R_{pub} \) runs already included execution time measurements for all relevant cache placements. Hence, variations are mostly caused by random variations in the execution time sample used. Note that this may lead to either higher or lower pWCET estimates. (2) In some cases, TAC accounts for cache placements likely not observed otherwise, such as for edn and jfdc. (3) Finally, in the case of ns, the pWCET estimate decreases by 15% with PUB+TAC w.r.t. PUB only. In this case we realize that execution times observed are already close to the maximum execution time possible in practice. Moreover, the application of TAC increases the number of runs from 3,000 to 500,000 to guarantee that high execution times are observed sufficiently. As a consequence, the set of highest execution times obtained with \( R_{pub+tac} \) is relatively more homogeneous, since an increasing number of runs makes execution times approach the maximum possible value asymptotically. As a result, MBPTA delivers a tighter bound and hence, the pWCET estimate decreases by 15%.

5 RELATED WORK

Several WCET analysis techniques have been proposed in the last decades [37], each of them with its pros and cons. While static timing analysis is still (and will continue to be) the preferred technique to deal with relatively simple high-critical systems, MBTA approaches are the most effective means to analyze cutting-edge systems, exploiting high-performance hardware features. Notably MBTA is the dominant practice in the automotive domain, and it has been successfully applied to the highest criticality software, e.g. DAL-A software in avionics [17], to meet functional safety standards requirements [12, 30]. MBTA is a probabilistic variant of MBTA that has been receiving increasing attention since early 2000’s [3, 8]. We stick to the standard notion of WCET (pWCET), identifying worst-case timing bounds for tasks in isolation, disregarding inter-task and inter-core effects: pWCET estimates computed with MBPTA hold valid by construction for multicores if hardware shared resources are MBPTA-compliant [35], or if contention in shared resources is accounted for a posteriori [7] and monitored during operation [25, 26].

The vast majority of works considering MBPTA build on top of EVT. Some of them analyze the implications of upper-bounding execution times with distributions potentially more pessimistic than exponential tails [19], using dependent data [31], and obtaining pWCET estimates in the presence of discrete and/or dependent data [18]. However, only few works analyze the relationship existing between the pWCET estimates obtained during analysis and the timing behavior during operation [2]. Guidelines for a correct application of MBPTA and potential issues are discussed in [13, 23, 32]. Statistical aspects in the application of EVT are explored in [13, 32] where EVT parameters selection and confidence intervals are the
main concerns. A more informed application of EVT to the software domain is advocated instead in [25], restricting EVT to using exponential tail distribution [2] on the account that it has been argued to be the most stable (and always overapproximating) distribution to model worst-case execution of real-time software [2, 27]. A universally recognized issue in the application of EVT is related to identifying an exhaustively representative input sample [13, 32, 38]; our work directly responds to this challenge by synthetically deriving a path-upperbounding version of the target program and defining the minimum number of observations to be collected on such version.

Achieving full path coverage in MBTA has been studied from different angles. Some works attempt to reduce programs to a single execution path building on constant-time predicated instructions [28], but its requirements and cost are difficult to leverage with industrial practice. Other approaches, instead, use genetic algorithms and model checking to generate input vectors to achieve full path coverage [4, 36], but confidence on the results is limited due to their heuristic nature. In the context of MBPTA, only PUB [14] and EPC [38] have been proposed to achieve full path coverage. PUB, upon which this work builds, modifies the source code to produce a program whose timing upper-bounds that of the original program. EPC, instead, operates on fine-grained measurements at basic block level to discount the benefits of specific path traversals and obtain path-independent measurements.

Some MBPTA works derive the minimum number of runs to obtain representative pWCET estimates. This concern was pointed out in [1, 29] and conveniently addressed in [1, 21] with some limitations related to the access sequences. Extensions to arbitrarily complex access sequences have been presented with limiting computational cost first [22], and with affordable costs later, as in the case of TAC [24]. However, to the best of our knowledge, our paper is the first attempt to simultaneously attack the full path coverage and representativeness problems in the context of MBPTA.

6 CONCLUSIONS
In this paper we tackle the issues of achieving full path coverage and representativeness against potential cache layout scenarios that may appear with low-probability. We demonstrate that both objectives can be achieved simultaneously, supporting our proposed method with empirical evidence. We show that the execution time distribution of any path of a pubbed program upper-bounds the execution time distributions of all paths in the original program. Then, we also show that by collecting a sufficient number of execution time measurements of any pubbed path, the pWCET estimate upper-bounds all paths in the original program under all cache layouts occurring with relevant probabilities. As a result we simultaneously deliver both full path coverage and cache representativeness.

ACKNOWLEDGMENTS
This work has been partially funded by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence. The Ministry of Economy and Competitiveness partially supported Suzana Milutinovic under FPI grant (BES-2016-077561), Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717) and Enrico Mezzetti under Juan de la Cierva-Incorporación postdoctoral fellowship (JCI-2016-27396).

REFERENCES