Understanding MBPTA and its requirements on program instructions (v2)

Jaume Abella[1], Eduardo Quiñones[1], Tullio Vardanega[2], Francisco J. Cazorla[1,3]
[1] Barcelona Supercomputing Center, Spain.
[2] University of Padua, Italy.
[3]IIIA-CSIC, Spain.

The research leading to these results has been funded by the European Community's Seventh Framework Programme [FP7/2007-2013] under the PROXIMA Project (www.proxima-project.eu) under grant agreement 611085.

Abstract

This white paper provides an understanding on the main requirements that Measurement-Based Probabilistic Timing Analysis (MBPTA) places on the hardware/software platform under consideration, the essence of which is to ensure that each program instruction can be attached an Execution Time Profile that describes the probabilistic distribution function of the execution time cost of that instruction. This document builds upon previously published papers with which the reader is assumed to be familiar. In particular, it builds upon [1], [3], [6].

This document is an extended version of the original one, from now on referred to as v1. While the technical content is exactly the same, some changes have been done in terms of presentation:

- In order to prevent misunderstanding among SPTA and MBPTA, when explaining how to combine the ETPs of different resources for MBPTA, we will not use convolution, but define new operators for the sequential and parallel composition of ETPs. This is so because convolution is used in the context of SPTA and requires independence among the ETPs being convolved. Instead sequential and parallel composition allows certain dependences among ETPs. This change exclusively affects Section III.
- Further we homogenize the way ETPs are represented with respect to previous publications on the field.

I. Introduction

Probabilistic Timing Analysis (PTA) methods [1], [3], and in particular Measurement-Based Probabilistic Timing Analysis (MBPTA) [3], reduce the information required to accurately model the execution time of applications running in a safety-critical system deploying performance-improving hardware (e.g., cache memories, multi-level caches, etc.). PTA seeks to determine WCET estimates for arbitrarily low probabilities of exceedance, termed probabilistic WCET or pWCET. As a consequence, even if any pWCET may in principle be exceeded, this can only happen with a given probability, which can be determined at a level low enough for the application domain; for example, in the region of 10^{-15} per hour of operation, largely below the acceptable probability of random hardware failures in a safety critical real-time embedded systems (e.g., those failures must be below 10^{-8} per hour of operation for the highest safety level in automotive as dictated by ISO26262 [4]).

Under PTA, the latency of each execution component at a given level of granularity can be assigned a distinct probability of occurrence. This behaviour is described by a probabilistic Execution Time Profile (ETP for short), which is expressed by the pair: <timing vector; probability vector>. The timing vector

in the ETP of a component enumerates all the possible latencies that an execution of (a request in) that component may incur. For each latency in the timing vector, the probability vector lists the associated probability of occurrence. Hence, for execution component $\mathcal{C}_i$ we have $ETP(\mathcal{C}_i) = <\vec{t_i}, \vec{p_i}>$ where $\vec{t_i} = (t_i^1, t_i^2, ..., t_i^{N_i})$ and $\vec{p_i} = (p_i^1, p_i^2, ..., p_i^{N_i},)$, with $\sum_{j=1}^{N_i} p_i^j = 1$.

PTA can be applied in either a static (SPTA) [1] or measurement-based (MBPTA) [3] fashion. Though we focus on the latter since it is closer to industrial practice. During the *analysis stage* of the system MBPTA collects a number of observations of the event of interest: end-to-end measurement runs of the program. MBPTA uses a statistical method called Extreme Value Theory (EVT) [10][3] to compute pWCET estimates that are a trustworthy upper-bound of the execution time of the program during the *deployment stage* of the system. MBPTA has its own requirements, some of which are inherited by the fact that MBPTA employs EVT [2]. EVT requires that the observed execution times of program runs (of the same path) have a distinct probability of occurrence and can be modelled with independent random variables. MBPTA also causes those variables to be identically distributed, so that MBPTA treats i.i.d. random variables.

MBPTA places strong requirements on the processor hardware and on how observations (end-to-end execution times of the program) are taken so they can be correctly modelled with i.i.d random variables. In this paper we show how, by having ETP at the level of *dynamic instruction*[1], MBPTA requirements are fulfilled. We show how to design processor resources such that we can derive an ETP for that resource. Then we show how to combine the ETP of the processor resources used by instructions executed by the program under analysis. We show how the guaranteed probabilistically analysable timing behaviour of each processor resource and of the program components (also known as units of execution) that use them can be combined into a probabilistically analysable composite timing behaviour meeting the MBPTA requirements.

II. MBPTA REQUIREMENTS

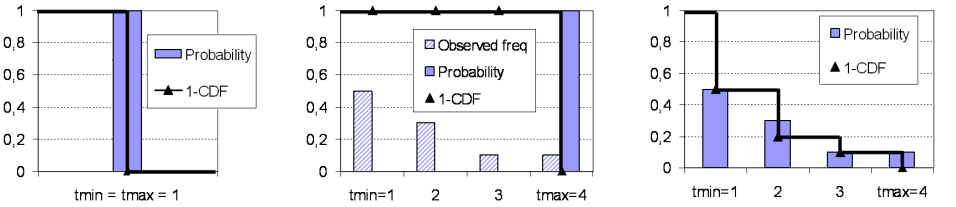
The primary goal of MBTA is to provide pWCET estimates that hold under execution conditions that may occur during actual operation: whereas those conditions may not be exactly identical to those captured by the observation runs made at analysis time, analysis time conditions must still reproduce or upper-bound the probabilities of the execution times that may occur during operation [2].

The fact that MBPTA uses EVT causes it to inherit EVT's own requirements. MBPTA then adds its own requirements to them, which serve to achieve its own goal [2]. Applying MBPTA requires to properly identify all the sources of execution time variation of the program under study. The nature of each source of variability has to be understood and properly captured at analysis time to guarantee that pWCET estimates hold during operation. This issue is analysed in this section, where we assume that the program for which a pWCET is to be estimated contains no system calls and that it is run in isolation. The interaction with other tasks and the operating system is taken into account during system integration as explained in [11][9]. First, we start assuming that the program contains a single path of execution. Later, in the next section we focus on multi-path programs.

A. Hardware resource types

We term *jitterless resources* those processor resources that have a fixed latency, independent of the input request and of the past history of service. Many hardware resources in current processor architectures can

[1] Any program instruction executed by the processor is a dynamic instruction, regardless of whether the instruction was fetched and executed before. For instance, if the program code has a *static instruction* I in a loop with M iterations, there will be M different dynamic instructions for I, say $I(1)$, $I(2)$,... $I(M)$. In fact, dynamic instruction designates an individual occurrence a given instruction in a given program path, whereas static instruction designates one of the processor instructions used in the program. In this white paper we use indistinctly instruction and dynamic instruction.



(a) Jitterless resource (b) Jittery upper-bounded resource (c) Jittery time-randomised resource

Fig. 1: Probabilistic timing behaviour of a single instruction for each type of resource

be classified as jitterless. Jitterless resources are easy to model for all types of static timing analysis. For PTA techniques, the ETP of a jitterless resource is given by: $ETP_{jl} =< (l), (1.0) >$, where l is the latency of the resource. In particular building the ETP of a simple instruction that uses a single resource requires knowing only whether the resource in question is jitterless (information implicit in the instruction) or whether the instruction is part of a sequence of instructions that must incur a delay when using a jitterless resource (information implicit in the architecture). With proper path and pipeline analysis, the types of the resources can be easily determined. Of course, measurements obtained from program runs that only use jitterless resources will perfectly capture their constant impact on execution time. The probability distribution function (PDF) and the complementary cumulative distribution function (1-CDF) of a jitterless resource are shown in Figure 1(a).

Other resources, for instance cache memories, have a variable latency: we call them *jittery resources*; their latency depends on their history of service, i.e. the execution history of the program, the input request, or a combination of both.

- Dependence on execution history. Some resources are stateful and their state is affected by the processing of requests. If latency depends on the internal state of the resource and this state is in turn affected by previous requests, then we say that the resource latency depends on the execution history of the program. With caches, the latency of an access request depends on whether the access is a hit or a miss, which in turn depends on the sequence of previous accesses to memory.
- Dependence on input request. In this case, the latency is determined by the data carried by the request. For a processor, these data are usually encoded in the instruction that issues the request, typically through input registers.

Jittery resources have an intrinsically variable impact on the WCET estimate for a given program. The significance of this impact depends on the magnitude of the jitter, the program under study, and the analysis method. For any given jittery resource, either all requests to it are assumed to incur the worst-case latency — as long as timing anomalies cannot occur [14] — or the resource is time-randomised. The particular designer's choice for a given resource needs to trade design cost for time-randomising resources against degradation of WCET estimate tightness for assuming the worst-case latency for some resources.

The ETP for resources assumed to worst-case latency can be expressed as $ETP_{wj} =< (l_{max}), (1.0) >$, where l_{max} is the worst-case latency of the resource. An example of such a resource with upper-bounded latency is shown in Figure 1(b). Note that actual probabilities for the different latencies are unknown and, at most, frequencies can be observed. Therefore, the ETP must be upper-bounded enforcing the highest latency at analysis time. Conversely, the ETP of a time-randomised jittery resource is: $ETP_{rj} =< (l_1, l_2 ..., l_k), (p_1, p_2, ...p_k) >$ where l_i and p_i represent the different latencies of the resource and their associated probabilities of occurrence. The PDF and 1-CDF for an example are shown in Figure 1(c).

B. Jitter propagators

The execution of an instruction in a time randomised resource leads to, potentially, new states each of which can be reached with a given probability. This means that the processor state existing prior to the execution of the instruction is transformed into a set of S new states, each one reachable with a given probability. For instance, in a time-randomised cache with N entries, on an eviction the new line can be allocated in any of the N entries of the cache with a given probability. That is, prior to the execution of the instruction i_j the cache has a given state CPS_{in} which after the execution of the instruction spawns into N states $exec(i_j, CPS_{in}) = \{CPS^1_{out}, CPS^2_{out}, ..., CPS^N_{out}\}$. The execution of multiple instructions using this kind of resources spawns each one of the reachable states into a number of new states each of which can be reached with a given probability.

The stalls that a *buffer resource* may introduce when full, affect the execution time of the program; as such they have to be considered in the analysis. Unlike caches however, which – in the PTA design – introduce probabilistic variability in the execution time and therefore generate new states in the probability tree, buffer stalls cannot produce probabilistic variability. Buffer latencies have *no* effect on the probability of each execution time to occur. Buffers may change the state of the processor, but they do it deterministically thus generating only one state based on any previous state. Therefore, their ETP is $ETP_{buf} =< l_{state} >, < 1.0 >$, where l_{state} can differ across different states. In that sense, buffers cannot create probabilistic jitter but simply propagate jitter or, in other words, given a sequence of outcomes for all probabilistic events the delay of the buffer resources is fully deterministic [5], making them analysable with MBPTA as measurements capture accurately their impact in execution time.

C. Assigning ETP to individual processor resources

In order for a resource to be treated with PTA, the probability assigned to each latency in the timing vector of the ETP for that resource must be a a probability in the strict mathematical sense. The probability for a given latency is different from its frequency. This is best shown with an example. Consider a resource R_1 with $\vec{t_1}= (t^1_1, t^2_1)$: latency t^1_1 in the timing vector would have a probability of occurrence $p^1_1 = 0.5$ if – in the implementation of that resource – on every request to it we flipped a coin and the request had latency t^1_1 if we saw heads and t^2_1 otherwise. In contrast, if for a deterministic stateful resource R_2 with latency $\vec{t_2}= (t^1_2, t^2_2)$ we *observed* that, for a given program, 50% of the requests take t^1_2 and 50% t^2_2 (e.g., it takes each latency alternatively), we would have a 50% observed frequency for each possible latency of that resource, but not a 50% probability. This is so because for events that are strictly dependent on the history of execution, cumulative information on past events *cannot* be used to provide guarantees about the appearance of future events.

For the purposes of PTA, the timing behaviour of jitterless and jittery (either upper-bounded or time-randomised) resources can all be described probabilistically by ETP. Figure 1 depicts an ETP for each such resource type. In each plot in the figure we show the relevant latencies of a request to a given resource type[2], where t_{min} and t_{max} stand for the minimum and maximum latency respectively. Latency values between 1 and 4 cycles have been used in this example.

Jitterless resources have an ETP with a single latency with probability 1 (vertical bar in Figure 1(a)). The Probability Distribution Function (PDF) described by the ETP has a single point at $t_{min} = t_{max} = 1$. Also the Complementary Cumulative Distribution Function or $1 - CDF$, also known as exceedance function, has a single point at t_{min} with value 0. For jitterless resources, the probability of latency $t > t_{max}$ is obviously 0.

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

Figure 1(b) depicts the probabilistic timing behaviour of an upper-bounded jittery resource. For these resources we just take their worst-case latency. The stripped bars in Figure 1(b) show the *observed frequencies* and not the actual probability of each latency. The solid bar shows the safe ETP assumed for resources of this kind. Assuming the worst-case latency may indeed cause pessimism in WCET estimation and thus also in our probabilistic bounding of it, but it also reduces implementation complexity. As for jitterless resources, the probability of latency $t > t_{max}$ is 0.

Figure 1(c) depicts the probabilistic timing behaviour of a time-randomised jittery resource. The solid bars show actual probabilities. In the figure, for example, we assume that the ETP of an exemplary time-randomised resource is $\{0.5, 0.3, 0.1, 0.1\}$ for $\{t_{min} = 1, 2, 3, t_{max} = 4\}$ respectively. Notably, ETP are discontinuous, which reflects the discrete nature of the latencies of the resources we consider. In general, there are no constraints on the latency l_i that a resource may have, other than $l_i \in \mathbb{N}$.

III. ETP OF INSTRUCTIONS

A composite ETP can easily be determined for every individual program component (ETP_{pc}), e.g. a dynamic instruction, that uses processor resources, which has an associated ETP describing their latency. That is $ETP_{pc} = f\left(ETP_1, ETP_2, ..., ETP_n\right)$, where ETP_i is the probabilistic execution time of resource r_i.

Sequential composition: sequential composition of ETP, $f_s\left(ETP_1, ETP_2, ..., ETP_n\right)$, leads to an ETP where latencies and probabilities are determined by the type of dependence across the input ETP, either systematic or probabilistic, as shown in next section. Sequential composition as intended here is architectural and not mathematical, hence different from the convolution used in the context of SPTA for combining ETPs of static instructions (e.g. instructions in the object code of the program).

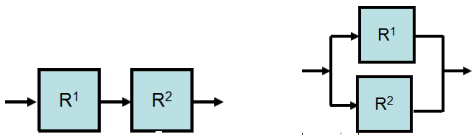
Let us assume two ETPs, $ETP_1 =< (1, 2), (0.5, 0.5) >$ and $ETP_2 =< (5, 10), (0.5, 0.5) >$. Further assume that whenever ETP_1 takes latency 1, then $ETP_2 =< (5, 10), (0.8, 0.2) >$ and whenever ETP_1 takes latency 2, then the second ETP is $ETP_2 =< (5, 10), (0.2, 0.8) >$. In this case, $ETP_{1+2} = f_s\left(ETP_1, ETP_2\right)$, leading to $ETP_{1+2} =< (6, 7, 11, 12), (0.4, 0.1, 0.1, 0.4) >$. Still, ETP_2 takes, for instance, latency 5 with probability 0.5 because $P(ETP_1 = 1) \times P(ETP_2 = 5) + P(ETP_1 = 2) \times P(ETP_2 = 5)$ is $0.5 \times 0.8 + 0.5 \times 0.2 = 0.5$.

The key appreciation here is that the dependence that ETP_2 has on ETP_1 can be modelled probabilistically as decribed in next section. As a result, the executions carried out during analysis, capture the behaviour of this dependence and hence, the pWCET estimate derived to bound the execution time during operation considers such dependence and upper-bounds its impact.

Parallel composition: processor resources may also be arranged in parallel. Examples of parallel resources are some particular designs of cache memories and TLB, where cache access and address translation can occur in parallel. With parallel arrangements, no dependence across ETP can exist, since for that to exist some sequential relation across ETP should occur, which should be addressed by sequential composition. The probabilities of the parallel composition ($f_p\left(ETP_1, ETP_2, ..., ETP_n\right)$) correspond to the multiplication of probabilities across ETP. However, the latencies correspond to the maximum latency of the probabilities multiplied.

This is illustrated with the following example. Let the ETP for two program components be $ETP_1 =< (1, 4), (0.4, 0.6) >$ and $ETP_2 =< (2, 3), (0.3, 0.7) >$ respectively. The ETP from their parallel composition, $ETP_{1+2} = f_p\left(ETP_1, ETP_2\right)$, is $ETP_{1+2} =< (2, 3, 4), (0.12, 0.28, 0.6) >$.

Whereas understanding what ETP stand for is an essential prerequisite to appreciate how PTA works, computing the ETP is *not* required for MBPTA.



(a) sequential resources (b) parallel resources

Fig. 2: Different resource arrangements

A. *ETP and i.i.d. properties*

If during the analysis stage, end-to-end measurements are taken starting from the same hardware and software state, for single-path programs, with fixed input data, then following the design principles presented in previous section ensures that an ETP for each dynamic instruction can be derived. Such ETP could be derived, for instance, generating all processor states after the execution of each instruction and attaching to each one its corresponding probability. Then, by considering all states immediately after the execution of the instruction of interest with their corresponding probabilities and the latency of the instruction in each case, we could determine its ETP.

This means that every dynamic instruction is attached one particular ETP. In other words, every time a dynamic instruction is executed, its latency has a probabilistic behavior captured by its ETP[3].

The ETP of an instruction describes its probabilistic execution time distribution. An ETP can be understood as a concrete representation of a random variable, where the different latencies that the instruction may take represent the possible outcomes of the random variable, and the probability of those latencies are the probability of occurrence of those outcomes.

If we make N executions of the program described above in a PTA-compliant architecture, the execution times observed for the (dynamic) instruction under consideration correspond to observations of the random event whose PDF is described by the instruction's ETP.

In this manner, independence is achieved because in each run of the program the initial hardware and software state is (reset to be) the same, whereby no dependence is retained with prior history. Identical distribution instead is obtained because the execution time of every execution of the instruction is described by the same ETP (which represents its probabilistic distribution function). We can therefore conclude that in a PTA-compliant architecture, collecting observations of measurement runs as described, dynamic instructions have an i.i.d. behaviour across all set of executions.

The property of independence and identical distribution can be erroneously construed as needing instructions, and their associated ETP, to be independent of one another. This is incorrect: the i.i.d. property applies – in certain conditions – to the observation of the execution time of individual dynamic instructions across multiple executions. Notably however, the i.i.d. properties may not apply across distinct dynamic instructions (that is to say, to fragments of program execution that contain more than one instruction). Instructions may in fact have dependences among them when the outcome of one random event that represents the execution of one dynamic instruction has an impact on the ETP of following instructions.

We call *causal dependence* any dependence among two instructions in a given precedence order such that the execution of the earlier one affects the timing behaviour of the later one. Obviously, the execution time of the earlier one determines when the later one can start executing, but our notion of causal dependence actually means that the latency a given instruction (the outcome of that random event) not only affects the time at which the later one starts but also its duration.

We differentiate two types of causal dependence among a source (preceding) instruction and a target (subsequent) instruction that do not prevent the latter instruction from exhibiting an i.i.d. timing behaviour

[3]Note that, by the definition given in footnote 1, a dynamic instruction is executed *once* per program run.

across program runs.

- *Systematic dependence*: The ETP of the target instruction is affected by the execution of the source instruction. This effect may alter the ETP of the target instruction in any way like, for instance, shifting some latencies in its ETP or making new latencies appear in the ETP of that instruction. None of this however causes the target instruction to lose its i.i.d. timing behaviour.
  This can be better understood with an example. Recall the goal of MBPTA is to control sources of execution time variability in such a way that the observations taken during the analysis stage can be used to upper bound probabilistically the timing behavior of the program during operation. Let us consider two instructions, one source and one target, on a given basic block[4], $bb1$. Let us also assume that the ETP of the target instruction is $ETP^{isol}_{target} =< (t_1, t_2, t_3), (p_1, p_2, p_2) >$ if it runs in isolation. Further assume that the execution of the target instruction as part of the basic block $bb1$, hence in the presence of the source instruction, is $ETP^{bb1}_{target} =< (t_1, t_2, t'_3, t_4), (p_1, p_2, p'_3, p_4) >$ in which the probability of t_3 changes and a new latency t_4 can be experienced. Given that the target instruction executed as part of the basic block, which in fact represents a dynamic instruction, is attached to a single ETP, ETP^{bb1}_{target}, irrespective of this being different from the ETP holding when the instruction was executed in isolation, and that this ETP stays constant during analysis and operation, it is the case that all observations of the execution time of the target instruction as part of this basic block are observations of this ETP^{bb1}_{target}. The key appreciation here is that the ETP must hold for every dynamic instruction over successive executions of the program. In the previous example, if the initial conditions are fixed, the target dynamic instruction in $bb1$ will have a single ETP.
- *Probabilistic dependence*: The execution of the source instruction has a probabilistic effect on the ETP of the target instruction. This is the case of memory accesses to a time randomised cache as explained in the following section. A probabilistic causal dependence causes that dynamic instruction to suffer a transformation in its ETP. However, given that the causal effect in the target instruction is probabilistic, this is equivalent to applying a transfer function $transf()$ that takes as an input an ETP and provides as an outcome another ETP $tranf(ETP^{isol}_{target}) = ETP^{bb}_{target}$. Again, the key appreciation is that the target (dynamic) instruction is always subject to the same ETP^{bb}_{target} thus enabling MBPTA to properly capture its timing effects at analysis time analogously as they will occur during operation.

Overall, on a PTA-compliant platform, any hardware and software state with bearing on the execution time after of any dynamic instruction of the program is reached with a given probability. Therefore, one can build the ETP *of every single program path* that can be traversed by an observable execution by collecting the execution time of each final state of that system and its corresponding probability of occurrence. Therefore, the execution time of the program as a whole (seen as the traversal of a given path) has an ETP and is, hence, a random variable with i.i.d. properties.

B. Time-randomised cache

In a processor, data caches are used by program memory instructions, while instruction caches are typically accessed by all instructions. If we assume a fixed latency to access all components with the only source of execution time variability being whether memory instructions hit or miss in the data cache, then the ETP of a memory operation can be described as: $ETP_{mem-op} =< (l_{hit}, l_{miss}), (p_{hit}, p_{miss}) >$. As stated before, i.i.d. properties are attained across program runs as needed by MBPTA, so the ETP_{mem-op} for a particular dynamic instruction describes its execution time probability distribution function.

[4]A basic block is a portion of the code with a single entry point and a single exit point.

The i.i.d. property could be achieved across instructions by a cache that on every access flushes all N lines in cache and it is reloaded with N lines randomly chosen from memory, which we assume has M lines with $M > N$. This would mean that the hit probability of every cache access is $\frac{N}{M}$ regardless of the execution history. However, this cache is both unrealistic and not needed for the application of MBPTA.

In reality we only need to ensure that memory operations that use the cache can be attached an ETP. This ETP can be dependent on execution history as long as that dependence is systematic or probabilistic as described in previous section. This requires that the events affecting the timing behaviour of the cache have a probabilistically modellable nature.

Let us consider the cache presented in [6] as an example of MBPTA-analysable cache. In the cited work, the authors present a set-associative cache that deploys random replacement, as used in many commercial processors, and a novel random placement policy and hardware implementation. In this white paper, we focus on a fully-associative cache version of the cache design proposed in [6] which only deploys random replacement (RR). A similar argumentation can be followed for set-associative caches deploying random replacement and random placement.

A RR policy ensures that every time a memory request misses in cache, a way in its corresponding cache set is randomly selected and evicted to make room for the new cache line. This ensures that (1) there is independence across evictions and (2) the probability of a cache line to be evicted is the same across evictions, i.e. for a W-way associative cache, the probability for any particular cache line to be evicted is $\frac{1}{W}$ for each set. In the particular case of a fully-associative (FA) cache, such probability holds for the only cache set.

Given a sequence of cache accesses, the ETP for each of them (i.e. its hit/miss probabilities) can be determined by computing how likely are previous accesses to evict the corresponding cache line. For instance, in the sequence <*A, B, C, A*>, B and C can evict A with a given probability that depends on the number of cache ways and whether B and C were fetched before or not.

Since cache lines evicted are chosen randomly, whether an access hits or misses depends solely on random events for a given sequence of accesses regardless of their absolute addresses, and thus the hit/miss outcome is probabilistic. In particular, starting from an empty cache state, the hit probability (P_{hit}) of a given access A_j in the sequence $< A_i, B_{i+1}, ..., B_{j-1}, A_j >$, where A_i and A_j correspond to accesses to the same cache line and each B_k accesses a different line (also different from cache line A), can be obtained as follows, with $P_{miss} = 1 - P_{hit}$ for any access:

$$P_{hit_{A_j}} = \left(\frac{W-1}{W}\right)^{\sum_{k=i+1}^{j-1} P_{miss_{B_k}}} \tag{1}$$

where $P_{hit_{A_j}}$ is the probability of A surviving all evictions performed by $< B_{i+1}, ..., B_{j-1} >$.

The probability of A to survive one random eviction is $\frac{W-1}{W}$. Meanwhile, given that one random eviction is performed on every miss, the total number of evictions equals the expected number of misses in between A_i and A_j, which is $\sum_{k=i+1}^{j-1} P_{miss_{B_k}}$. Using Equation (1) the hit/miss probabilities of each access can be derived sequentially starting from the first cache access. If no access has been performed to A before A_j, then the hit probability is zero.

Accuracy of Equation 1. Equation 1 is only fully accurate when each Bj accesses a different line and none of them accesses the same cache line accessed by Ai and Aj, subject to an initial state with an empty cache. The equation shows that the dependence that exists among a target memory operation accessing address Aj and all its source instructions accessing Bj can be captured probabilistically. As a result, the target memory operation has an associated ETP. The equation becomes an approximation when accesses

Bj repeat and/or the initial cache state is not empty. The accuracy of the approximation heavily depends on the reuse patterns of Bj and way size (W). We have verified empirically that approximations are in the range $[P_{hit}^{real} - 0.01, P_{hit}^{real} + 0.01]$ where P_{hit}^{real} is the actual value, for cache sizes resembling real ones in size ($W \geq 64$ entries) and meaningful programs (e.g., EEMBC autobench benchmark suite [13]) for most of the data and instruction cache accesses. Deviations can be larger for ridiculous sizes (e.g., $W = 2$ or $W = 4$). To the best of our knowledge no real processor deploys data or instruction caches of that size. However, this is irrelevant for MBPTA, since what really matters is the hit/miss outcome to have *a probability* rather than one particular value of that probability. As explained before, as long as the ETP exists at the level of dynamic instruction (and it does with time-randomised caches), the ETP exists at the level of end-to-end runs of the program, so the execution time of the program is an i.i.d. event and can be studied with MBPTA.

Equation 1 in the context of MBPTA. Having an ETP at the level of dynamic instructions allows the required i.i.d properties to emerge at the level of end-to-end measurement runs. MBPTA needs to collect a sufficiently large sample of such measurements – whose size is determined by the MBPTA method itself –, but does not need Equation 1 at all. As shown in [6], [8], [7], neither Equation 1 nor any other equations are used anywhere in the MBPTA method to obtain pWCET estimates. In fact, equations such as Eqn 1 are only provided as a means to illustrate the probabilistic nature of the events of interest. Still, those equations approximate quite well actual probabilities in the ETP for realistic cache sizes and realistic access patterns, so one might consider using them to estimate the expected number of misses in a program. However, those equations are neither upper- nor lower bounds for hit/miss probabilities. Hence, they cannot be used in the context of the static variant of PTA, called Static Probabilistic Timing Analysis (SPTA) [1].

C. Input Data Dependence under MBPTA

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 [1]. 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 suite 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 some particular integer values may take longer than other values, 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 plausibly required to supply worst-case data nor even statistically representative data for use with timing analysis; it is very unlikely that users know (in general) what the input data given to the program will actually be during operation, so we must design the platform in a way that causes no temporal effect to stem from input data. More details on this issue are provided in Section III-E below.

There are hardware proposals which force operations, such as divisions, to always take their upper-bound execution time, which is a safe (pessimistic) assumption [12]. In [12] the authors introduce a 'Worst-Case Mode', that simply delays responses for requests to a resource until its maximum execution time is elapsed. This feature is configurable by software and once activated it is transparent to the executing programs. For the purposes of this paper such a technique has to be present in the target architecture to upper bound jitter in some jittery resources whose timing cannot be randomised. It will be activated during the collection of execution times and can be switched off during operation.

A similar solution can be applied to input-data-dependent memory accesses, which must be flagged so that effects due to the input data can be accounted for as described in [3]. Most current Instruction Set Architectures (ISA) include some hint bits in the operation code of memory operations, which are used, among 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 will have a single use, it can set 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 flagged by means of these hint bits to prevent cache or other performance-improving features being used for these instructions which, otherwise, could have ETP at analysis time not upper-bounding those at deployment.

D. Multipath programs

Real-world programs have different execution paths. Current MBPTA requires the user to be able to identify the program paths that can lead to large execution times and to supply the input data needed to exercise them. As the i.i.d. properties that must hold for end-to-end runs effectively apply to individual program paths, MBPTA uses observations from *all* of the exercised paths during analysis, to feed EVT and provide pWCET estimates that hold valid for all of those paths, regardless of their execution frequency during operation. In essence, MBPTA generates a pWCET distribution that upper bounds all the observed executions.

How to identify the program paths of interest is not yet supported by MBPTA. Moreover, the path enumeration techniques used by classic static timing analysis do not directly fit the bill because of the different perspective that they have on the notion of worst case from what MBPTA needs. The crux of the difference is that time-randomised caches (and any time-randomised resource in fact) naturally reduce abrupt performance variations caused by minor changes in the execution conditions in comparison to deterministic, state-dependent resources (such as modulo placement and LRU replacement caches). This observation arguably helps the user identify the program paths of interest, as they are more likely to be within the longest ones as opposed to anywhere as it can be with non-PTA processor architectures. For instance, whether libraries or the operating system are placed at different locations at analysis and operation time can cause abrupt performance variations with time-deterministic caches, but it does not with time-randomised ones. Still, the identification of the program paths that are relevant for pWCET estimation is currently left to the user: with present MBPTA it can be only claimed that the pWCET estimate obtained is valid for the program paths that were exercised by the input data provided by the user. Interestingly, the viability of current MBPTA considerably increases in application domains where the development standards in place require functional testing to achieve high, where not full path coverage. Even more so when one considers that there is an obvious correlation between the test runs needed for functional testing and the measurement runs needed for MBPTA observation: in contract with other timing analysis techniques, MBPTA applied in those domains does not require the user to carry out additional ad-hoc activities on top of their standard practices.

E. Randomising Input Data

In theory, it would be possible to sample input data for the program from the universe of input data of the system in operation, so that the sample of measurement runs provided to MBPTA was meaningful for any input path. In fact, one could even perform such sampling considering all combinations of sources of execution time variation (SETV) – including execution paths –, run experiments on a time-deterministic architecture and apply EVT. However, as explained in [2] one cannot expect the user to be able to enumerate all input data that will be exercised during actual operation, by all units of the system, and

also provide their frequencies. In fact, it would also be acceptable if the user was able to, at least, enumerate those combinations of SETV but, in general, we cannot expect the user to be able to provide representativeness for those units with input-dependent latencies, for the placement in memory of all objects, for the execution paths, etc., including all of their combinations during analysis. Therefore, by upper bounding and randomising the timing of a number of processor resources, the user is not bound to guarantee the representativeness of the test data for many of those SETV. We can therefore maintain that PTA-compliant platforms used in conjunction with MBPTA allow the user obtaining meaningful pWCET estimates with much less burden on the user than conventional MBTA approaches on top of non-PTA-compliant platforms.

IV. CONCLUSIONS

For MBPTA ETP for each instruction need not be determined so long as they are known to exist at some level of granularity in the execution stack *and* their i.i.d. nature is propagated all the way up to the observed events: this will be sufficient for the observations to reflect the effect of actual probabilities of each resource for each request.

REFERENCES

[1] F.J. Cazorla, E. Qui nones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. Proartis: Probabilistically analysable real-time systems. *ACM TECS*, 2013.

[2] F.J. Cazorla, T. Vardanega, E. Quinones, and J. Abella. Upper-bounding program execution time with extreme value theory. In *WCET Workshop*, 2013.

[3] L. Cucu-Grosjean, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Quinones, and F. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, pages 91–101. IEEE, July 2012.

[4] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.

[5] L. Kosmidis, T. Vardanega, J. Abella, E. Quinones, and F.J. Cazorla. Applying measurement-based probabilistic timing analysis to buffer resources. In *WCET workshop*, 2013.

[6] Leonidas Kosmidis, Jaume Abella, Eduardo Quinones, and Francisco J. Cazorla. A cache design for probabilistically analysable real-time systems. In *DATE*, 2013.

[7] Leonidas Kosmidis, Jaume Abella, Eduardo Quinones, and Francisco J. Cazorla. Multi-level unified caches for probabilistically time analysable real-time systems. In *RTSS*, 2013.

[8] Leonidas Kosmidis, Jaume Abella, Eduardo Quinones, and Francisco J. Cazorla. Efficient cache designs for probabilistically analysable real-time systems. *IEEE Transactions on Computers*, to appear.

[9] Leonidas Kosmidis, Eduardo Quinones, Jaume Abella, Tullio Vardanega, and Francisco J. Cazorla. Achieving timing composability with probabilistic timing analysis. In *ISORC*, 2013.

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

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

[12] Marco Paolieri, Eduardo Quinones, Francisco J. Cazorla, Guillem Bernat, and Mateo Valero. Hardware support for WCET analysis of hard real-time multicore systems. In *ISCA*, Austin, TX, USA, 2009.

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

[14] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A definition and classification of timing anomalies. *Int'l Workshop On Worst-Case Execution Time Analysis (WCET 2006)*, 2006.