EPC: Extended Path Coverage for Measurement-based Probabilistic Timing Analysis

Marco Ziccardi, Enrico Mezzetti and Tullio Vardanega
Department of Mathematics
University of Padua, Italy

Jaume Abella and Francisco J. Cazorla*
Barcelona Supercomputing Center, Spain
*Spanish National Research Council (IIIA-CSIC)

***Abstract*—Measurement-based probabilistic timing analysis (MBPTA) computes trustworthy upper bounds to the execution time of software programs. MBPTA has the connotation, typical of measurement-based techniques, that the bounds computed with it only relate to what is observed in actual program traversals, which may not include the effective worst-case phenomena. To overcome this limitation, we propose Extended Path Coverage (EPC), a novel technique that allows extending the representativeness of the bounds computed by MBPTA. We make the observation data *probabilistically path-independent* by modifying the probability distribution of the observed timing behaviour so as to negatively compensate for any benefits that a basic block may draw from a path leading to it. This enables the derivation of trustworthy upper bounds to the probabilistic execution time of *all* paths in the program, even when the user-provided input vectors do not exercise the worst-case path. Our results confirm that using MBPTA with EPC produces fully trustworthy upper bounds with competitively small overestimation in comparison to state-of-the-art MBPTA techniques.**

I. Introduction

Determining trustworthy and tight bounds to the worst-case execution time (WCET) of software programs as a means to guarantee that the system timing constraints are met is a hard problem in general. Several timing analysis techniques have been proposed in the past [1] to cope with the WCET estimation problem. Static timing analysis (STA) relies on the creation of an accurate model of the hardware and of the software running on it, to derive a WCET bound for a program without executing it. Measurement-based timing analysis (MBTA) instead aims at deriving realistic bounds directly from on-target measurements of end-to-end runs of the program of interest, performed over a user-provided subset of all possible input vectors. More advanced *hybrid* MBTA approaches collect measurements on smaller parts of the program and conservatively combine them to produce more trustworthy WCET estimates. However, the trustworthiness of STA and standard/hybrid MBTA approaches is challenged by threats related with the models or the testing procedures they use [2]. In particular, MBTA techniques cannot be proven in general to provide fully trustworthy upper bounds to the WCET, unless all possible program paths and execution conditions have been exhaustively observed. Although different levels of testing are typically required for functional verification (e.g. modified condition/decision coverage in DAL A functions under DO-178C [3]), the exhaustive path coverage required by MBTA is not attainable in the general case.

Measurement-Based Probabilistic Timing Analysis (MBPTA) [4] has recently been introduced as a means to overcome some of the limitations of both STA and MBTA. In contrast with STA, MBPTA computes WCET bounds without relying on the construction of abstract models of the system. Unlike MBTA, MBPTA is able to attach quantitative confidence to the obtained results. MBPTA computes a probabilistic WCET (pWCET) for the program under analysis: the pWCET represents a bound to the program's execution time that can be exceeded only with a given probability, determined in accord with system needs (e.g., 10^{-15} per activation). pWCET estimates computed with MBPTA are valid only for the paths that have been exercised by the input vectors as provided by the user [4]. This limitation has recently been addressed by the Path Upper-Bounding (PUB) [5] technique, which allows MBPTA to derive a trustworthy pWCET for a program even when one cannot guarantee that all paths have been traversed in the observation runs. PUB achieves this by collecting measurements on an extended version of the target program, where all conditional constructs are modified to exhibit a probabilistic timing behaviour that upper-bounds all possible alternative branches. To this end, PUB postulates the availability of the program sources, and of (qualified) technology to apply semantic-preserving modifications. Both assumptions are admittedly difficult to meet in practice.

In this paper we present a new technique, named Extended Path Coverage (EPC), which extends the standard MBPTA process and enables the computation of a pWCET for a certain exceedance probability that is valid for the whole program, without relying on either full path coverage or the provision of the worst-case inputs. As a distinctive feature over PUB, EPC only relies on measurements over the original program. EPC builds on the concept of *probabilistic path-independence* to derive a collection of execution times that is representative of all program paths, just relying on the availability of a set of measurements for each basic block (hence requiring *basic block coverage*), irrespective of the path leading to it. Notably, basic block coverage represents a common and relatively lightweight coverage requirement, required by DO-178C [3] already from DAL C. The standard MBPTA process benefits from the path-independent observations generated by EPC, in ensuring the same degree of trustworthiness as achievable with full path coverage. Our results show that the pWCET computed with EPC produces fully trustworthy upper bounds to the program execution time while incurring only a 12% average increase over plain MBPTA on selected paths. Thus, EPC allows computing bounds that are on average tighter than those obtained with MBPTA alone plus the traditional industrial engineering margin – often set to 20% or higher [6].

The remainder of this paper is organized as follows: Section II provides background on MBPTA and sets our basic assumptions; Section III introduces EPC and describes the overall process. Section IV focuses on probabilistic path independence and how it can be actually enforced. Section V discusses how EPC guarantees synthetic full path-coverage. An experimental evaluation of our approach is given in Section VI. Finally, Section VII surveys related work, and Section VIII draws some conclusions.

II. Background and assumptions

MBPTA builds on the application of Extreme Value Theory (EVT) [7] to a set of execution time observations, typically end-to-end measurements, taken on a software program. MBPTA allows computing an upper bound to the probability that the execution time of a given program on a given platform may exceed a given value. MBPTA users are interested in the pWCET value whose exceedance probability is no greater than a domain-specific threshold[1], possibly related to failure rates at system level. pWCET values are not attached to specific paths in the program under analysis but are valid for the whole program, which differentiates MBPTA from other probabilistic approaches [8]. EPC relates to MBPTA as a meliorative technique specifically aimed at improving the trustworthiness of its results. In particular, EPC is an unintrusive approach that operates on the observations provided in input to the MBPTA process, with no bearing on its specific implementation. Yet, the fact that EPC is deployed on top of the MBPTA process causes it to inherit some MBPTA assumptions.

State-of-the-art MBPTA approaches rely on two prerequisites. A first necessary condition emanates from basic EVT and establishes that the execution times of observation runs of the program under analysis must be modellable with *independent* and *identically distributed* (i.i.d.) random variables. This requirement has been shown to be achievable via the adoption of time-randomized hardware [9], [10]. In this work we focus on single core systems and consider time-randomized architectures as key enablers to the use of MBPTA. Time-randomized architectures are characterized by hardware resources whose timing behaviour is either randomized [9] or enforced to operate in a worst-case mode at analysis time [11]. Notably, time-randomized architectures are not a conceit: concrete FPGA implementations are being finalized [12]. Time-randomized architectures typically adopt time-randomized caches [13], with random policies for placement and replacement. For random placement (RP) – in set-associative caches – the address-to-set mapping is defined on the basis of a randomly generated seed, where the same seed is used in each run, but re-generated across runs. For random replacement (RR), we assume our caches to adopt an *evict-on-miss* policy, which (for a given cache set) randomly selects the cache line to be evicted in the event of a miss. At core level, we assume a simple pipelined processor, similar to the LEON 4 processor [14], in which each instruction incurs a fixed latency, and the core hardware units (e.g., IU, FPU) are fully pipelined, thus avoiding dependence between subsequent instructions. This kind of hardware architecture has been shown to be analysable with MBPTA techniques [13].

A second prerequisite of MBPTA is related to the *trustworthiness* of pWCET results, which directly depends on the representativeness of the observations fed into the analysis process [15]: measurements taken during the analysis phase are required to capture all factors of execution time variability that may stem from the hardware resources used in the system. The latter, however, may not always be a trivial task. With time-randomized caches, for example, the probability of some random placement events may be so low to be hard to capture with a few (thousands of) runs [16], [17]. As shown in [17], [18] these rare events can be detected and cured with additional observations. This notion manifests how critical it is to determine the minimum number of runs required to capture execution time variability. A simple *convergence criterion* to determine the minimum number of runs required, for example, to fully characterize the effects of time-randomized caches on the execution time of a given program path is given in [4]. In this paper we assume the use of that criterion or an equivalent mechanism. However, regardless of the amount of measurement collected, the pWCET estimates computed by current state-of-the-art MBPTA approaches only upper-bound the timing phenomena that have been exercised by the input vectors provided by the user [4]. With EPC we want to lift this limitation. EPC addresses the lack of trustworthiness resulting from non-exhaustive path coverage, by synthetically extending the set of actual measurements. In this paper we focus on the effects of time-randomized caches, as we deem them to be the main sources of variability in time-randomized architectures. The effects of core-level dependence among instructions are excluded by construction in the processor considered in this work.

We refer the interested reader to the original works on MBPTA for details on the timing analysis process, including: properties needed from the execution platform [4], [10]; measurement collection process [15]; handling of program data [10] and control dependence [5], collectively referred to as structural dependence, which is notably different from the statistical independence needed by MBPTA.

III. The Extended Path Coverage Approach

The inherent limitation of measurement-based approaches is that their bounds are only valid for the set of paths and execution conditions for which the available observations were collected [15]. MBPTA does not differ as it allows the user to probabilistically capture the sources of variability that stem from history of execution but still it only computes an upper bound to the timing behaviour observed in the traversed execution paths. To overcome this limitation, EPC *synthetically extends* the set of observations to obtain the equivalent effect of full path coverage. EPC's only requirement consists in a set of execution time observations over all program's basic blocks – the smallest units of execution comprised of strictly sequential code. Collecting time measurements at the granularity of basic blocks is an industrially viable option: the overhead incurred by probing the executions can be negligible with the adoption of advanced hardware debug interfaces or trace tools [19]. Industrial quality tool support to this end exists [20].

EPC builds on a twofold intuition. First, probabilistic execution times for a basic block can be made path-independent: probabilistic execution times in EPC are probabilistically augmented (or *padded*) to negatively compensate for the benefits that a basic block may draw from a specific traversal path leading to it, owing to sensitivity to history of execution. Second, we observe that probabilistically path-independent execution times of each basic block can be used as building elements in the construction of a collection of end-to-end execution times representative of *all* program paths, including those not directly exercised by the user-provided input vectors. Path-independence is an absolute need as plain execution time observations over basic blocks (even maxima values) cannot be soundly combined to obtain the execution time of unobserved paths. In fact, each observation is only relative to the particular path captured in the run, owing to cache-level dependence effects (the main focus of this work)

[1]For example, airborne applications at DAL-A prescribe a maximum failure rate of 10^{-9} per hour of operation, where a WCET overrun can be equated to a failure.

and core-level dependence effects, which are avoided in the processor we consider. EPC specifically copes with these dependencies by design, by characterizing the probabilistic impact of unobserved paths on the set of observed execution times. Finally, the artificially extended set of execution times generated by EPC for the whole program can be fed into the normal MBPTA process. By reason of these extended measurements, the representativeness of the computed pWCET is substantially improved and can be regarded as a fully trustworthy upper bound to the pWCET that would be obtained by exhaustively observing every possible path.

A. EPC basic concepts and notation

EPC builds on the notions of probabilistic execution and execution time profiles. Both concepts require a precise definition as they mark the difference with deterministic timing analysis.

Definition (Execution Time Profile)**.** *The Execution Time Profile (ETP) of a program or part thereof is the discrete probability distribution function that describes the execution times that the unit under analysis may exhibit.*

From the ETP we can derive the probability of a particular execution time value:

Definition (Probabilistic Execution Time)**.** *The Probabilistic Execution Time pET of a program or part thereof is defined as the probability that the execution time of the unit under analysis is equal to a given value.*

ETPs constructed from empirical observations, rather than statically determined a priori, are termed *empirical* ETPs.

Definition (Empirical Execution Time Profile)**.** *The Empirical Execution Time Profile (EETP) of a program or part thereof is the ETP of the unit under analysis, as determined by a set of observations.*

EPC operates on a set of $\mathrm{EETP}(bb_i)$ for all basic blocks in the program, as derived from actual observations $Obs(bb_i, \phi)$ over a finite subset[2] of all possible paths. Hence $\mathrm{EETP}(bb_i) = \{Obs_1(bb_i, \phi), Obs_2(bb_i, \phi'), \ldots, Obs_k(bb_i, \phi'')\}$. $\mathrm{EETP}(bb_i)$ can be made *probabilistically path-independent* by removing the positive effects of a specific execution path. This is obtained by augmenting each observation in $\mathrm{EETP}(bb_i)$ with the application of an execution time penalty or padding – $pad(bb_i)$ – according to a given probability distribution. We refer to the augmented observation over bb_i as $Obs^+(bb_i)$[3], which in turn contributes to an augmented empirical execution time profile, $\mathrm{EETP}^+(bb_i)$, with $\mathrm{EETP}^+(bb_i) = \{Obs_1^+(bb_i), Obs_2^+(bb_i), \ldots, Obs_k^+(bb_i)\}$. Padding is applied following a probability distribution that guarantees that $\mathrm{EETP}^+(bb_i)$ safely over-approximates the worst execution time distribution for bb_i across *any possible* program path that traverses it. The definitions of EETP and Obs can also be extended to end-to-end paths, as sequences of basic blocks. The central trait of EPC is that it constructs synthetic execution time observations and profiles for a path ϕ_i, denoted $\widehat{Obs}(\phi_i)$ and $\widehat{\mathrm{EETP}}(\phi_i)$ respectively, by combining the probabilistically augmented execution times for each basic block in the path.

[2]Determined, for example, by user-defined control-flow constraints.

[3]Path information are dropped in Obs^+ that is now path independent.

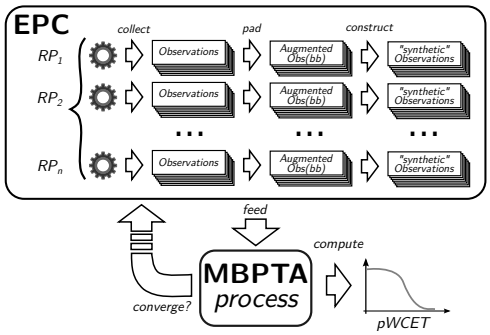


Fig. 1: EPC interaction with standard MBPTA process, where RP_i stands for the i-th random placement.

B. Overall approach

In randomized processor architectures, time-randomized caches are the main source of execution-time variability. Given a cache placement, the execution time of bb_i depends on the cache state, which in turn is affected by both the RR policy and the traversed execution path (i.e., history of execution). The effects of RP and RR are normally taken into account by the MBPTA analysis framework where convergence criteria guarantee that the observations given in input to EVT do convey the whole spectrum of those phenomena. Our approach, instead, is meant to improve the MBPTA approach by specifically dealing with the (otherwise uncontrolled) path-related variability. To this end, EPC uses the concept of *path independence*. Path-independent execution times, in fact, abstract away from specific execution paths and serve as a valid upper-bound for any execution context. A more formal definition of path independence follows.

Definition (Path independence)**.** *The execution time of a basic block bb_i in a program $\mathcal{P}$ is path-independent if its value does not depend on a particular path in $\mathcal{P}$. Accordingly, a set of execution times is path-independent if it includes path-independent execution times only.*

In EPC, we resort to *probabilistic path independence* to exploit the probabilistic nature of both MBPTA and time-randomized caches. Probabilistic path independence is enforced by adding a *probabilistic padding* to negatively compensate potential positive effects of variability (e.g., a cache hit) owing to a specific path. The probabilistic connotation stems from the fact that an execution time penalty (scalar) is computed following a probability distribution. However, in the computation of padding for basic blocks, the different sources of variability (RP, RR and execution path) cannot easily be told apart: the effects of RP along a specific path also depend on the specific cache placement being used. A holistic approach to path independence would involve the computation of a padding to counter the positive effects of both RP and RR. Yet, conservatively accounting for RP in the computation of path-independent execution times for each basic block would incur untenable pessimism.

For this reason, we isolate the effects of RP on the cache state and compute a padding that accounts for the effects of RR only. Observations, EETPs and paddings are thus relative to a specific cache placement. The effect of cache placement is handled within the standard MBPTA process, where execution times are collected over a set of randomly-generated cache placements so that the obtained pWCET is representative of the possible cache placement at deployment time. Hence, the application of EPC is only relative to a given cache placement

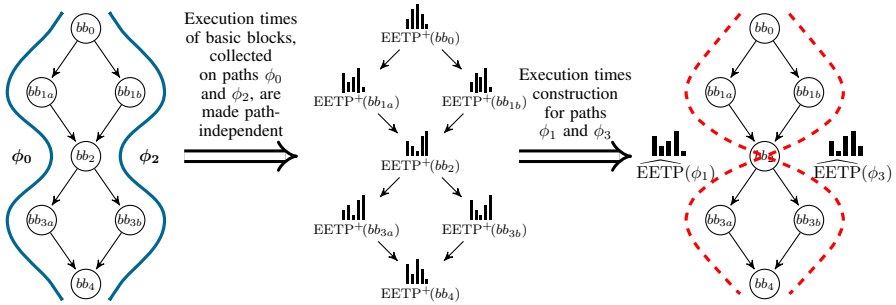


Fig. 2: Application of the main steps of EPC (repeated separately for each random placement) on a simple program comprised of two cascading conditional constructs. First, execution times for individual basic blocks are collected across paths ϕ_0 and ϕ_2. Subsequently, the execution times collected for each basic block are augmented to make them path-independent. Finally, execution-time values are synthetically computed for each non-observed path, ϕ_1 and ϕ_3.

and is to be repeated for any RP to be considered in the encompassing MBPTA process.

Figure 1 shows the interplay between EPC and standard MBPTA. The EPC process consists of three main steps: (i) *execution time collection*, (ii) *basic block padding* and (iii) *execution time construction.* All steps are specific to a given cache placement as each placement may determine different path-independent execution times. The original end-to-end observations and the synthetic ones, constructed by EPC, are finally given in input to the MBPTA to compute a trustworthy pWCET, for any exceedance probability, that is representative of all paths in the program $\mathcal{P}$ and all possible address-to-set mappings. Further observations or synthetic measurements may be required to meet the MBPTA convergence criterion.

The core steps in EPC are exemplified in Figure 2. In the first step we collect a number of end-to-end execution times for $\mathcal{P}$ on a finite and not necessarily exhaustive set of paths. Timing information for each basic block is then extracted from the collected execution times. This enables the computation of an empirical execution time profile $\text{EETP}(bb_i)$ for each bb_i under the current random placement[4]. The second step applies to all basic block observations $Obs(bb_i, \phi)$ that form an $\text{EETP}(bb_i)$. By adding a probabilistic padding, each $Obs(bb_i, \phi)$ is made probabilistically path-independent, denoted $Obs^+(bb_i)$. Augmented observations define an augmented execution time profile $\text{EETP}^+(bb_i)$ that is independent of the observed paths, and in fact over-approximates the timing behaviour of bb_i along *all* paths in the program that traverse it. Finally, the last step in the EPC process consists in combining path-independent $\text{EETP}^+(bb_i)$s to build a synthetic execution time profile $\widehat{\text{EETP}}(\phi)$ for each non-observed[5] path ϕ. By using probabilistically path-independent $\text{EETP}^+(bb_i)$, we ensure that the building blocks in the computation of an $\widehat{Obs}(\phi_i)$ are always valid upper bounds, as they hold for any program path ϕ_i. This in turn guarantees that by feeding a complementary set of $\widehat{\text{EETP}}(\phi)$ to MBPTA we obtain a pWCET that is valid for *all* execution paths, for any exceedance probability. The application of EPC ends by grouping all the artificially-constructed end-to-end execution times obtained under different randomly-generated cache placements, and feeding them to MBPTA.

We now discuss each EPC step in isolation. Execution time collection is a simple task that does not pose significant challenges other than concerns of industrial viability, which we already addressed. Section IV presents a technique for computing probabilistically path-independent execution times for each basic block, whereas Section V discusses a method for combining them to build synthetic observations.

[4]We avoid using oppressive notation by not including an index to represent the cache placement.

[5]Observed paths already have a valid EETP.

IV. Computing path-independent execution times

Path independence is a well-known property in static analysis approaches where it is not unusual to make conservative assumptions (e.g., assuming a cache miss when no better information is available). We argue, however, that path independence is also a property that can be enforced *a posteriori* in dynamic analysis approaches based on execution time observations. In this work we leverage the probabilistic framework provided by time-randomized architectures – presented in Section II – to make observations *probabilistically* path-independent, accounting for a conservative probabilistic time penalty (*padding*) in the observed execution times of each basic block. We rely on a limited amount of structural information, such as program control flow graph, loop structures and memory accesses. In fact, we do not need to know the exact accessed addresses but only whether any two addresses are mapped to the same cache line. This information can be typically obtained off the compilation process or collected by means of advanced debug interfaces [19]. In the following we show how to compute what we call a *probabilistic padding*.

A. Probabilistic padding

For each bb_i we apply a padding $pad(bb_i)$ to each observation, leading to path-independent observations. Focusing on cache-level dependence effects, $pad(bb_i)$ is a compound factor:

$$Obs^+(bb_i) = Obs(bb_i, \phi) + \sum_{@_I \in bb_i} pad^I(@_I) + \sum_{@_D \in bb_i} pad^D(@_D)$$

where a penalty is added for each memory access $@_A$ in bb_i, which can be either an instruction or data fetch (denoted $@_I$ and $@_D$ respectively). The naive approach to enforce path independence would consist in resorting to static scalar (i.e., non-probabilistic) padding. As a simple example, a pessimistic $pad^D(@_D)$ for a data access $@_D$ in bb_i is the difference between a miss and hit latency $L_{miss} - L_{hit}$ as this penalty would compensate for the fact that during an observation over bb_i we may have found $@_D$ already in the cache. Statically computing a padding, as was done in the previous example, is overly pessimistic, even worse than assuming all misses.

Unnecessary pessimism can be avoided by computing (and applying) a *probabilistic padding*, as opposed to the static one, which can be described according to a probability distribution. Intuitively, this observation is motivated by the fact that, within

a probabilistic framework, we do not need to enforce each observation Obs^+ to upper-bound the worst-case behaviour: we only need to ensure that the resulting distribution EETP^+ over-approximates the worst execution time distribution for that basic block across all program paths that traverse it.

The random nature of time-randomized caches causes the fact that for each path ϕ leading to a basic block *bb*, and for each memory address A accessed within *bb*, the latency incurred on access to A (i.e., $@_A$) follows a probability distribution, which we call Access Time Profile (ATP).

$$ATP(@_A, \phi) = \left\langle \begin{array}{cc} L_{hit} & L_{miss} \\ P_{hit}(@_A, \phi) & P_{miss}(@_A, \phi) \end{array} \right\rangle$$

Along each path, the ATP is determined by whether the intermediate accesses between the current and the previous access to $@_A$ hit or miss in cache, which in turn is determined by the random nature of RR. Hence, since an ATP captures the variability stemming from RR along a given path, we can define a path-independent ATP for $@_A$ that "upper-bounds" all $ATP(@_A, \phi)$, for all ϕ in the program. Note that upper-bounding ATPs is a *well-formed* concept as long as a *partial order* over ATPs can be defined.

Definition (Partial order over ATPs)**.** *Let ATP_1 and ATP_2 be two discrete execution time profiles for an access $@_A$. We say that ATP_1 upper-bounds ATP_2 ($ATP_1 \succeq ATP_2$) if, for each execution-time value in ATP_2 the cumulative probability obtained from ATP_1 is less or equal to the cumulative probability obtained from ATP_2 for that value.*

The *exact* path-independent $\overline{ATP}(@_A)$ is precisely determined by the worst-case miss probability of $@_A$. If $\overline{P_{miss}}(@_A)$ is an upper-bound to the miss probability for $@_A$ along any possible path then the complementary lower bound to the hit probability is $\overline{P_{hit}}(@_A) = 1 - \overline{P_{miss}}(@_A)$ and:

$$\overline{ATP}(@_A) = \left\langle \begin{array}{cc} L_{hit} & L_{miss} \\ \overline{P_{hit}}(@_A) & \overline{P_{miss}}(@_A) \end{array} \right\rangle$$

By construction $\overline{ATP}(@_A) \succeq ATP(@_A, \phi)$, $\forall \phi \in \mathcal{P}$ (by definition of $\overline{P_{hit}}$). Augmenting each *Obs* in $\text{EETP}(bb_i)$ according to the so-computed ATP upper bounds would produce our sought path-independent Obs^+.

Unfortunately, there are no practical means to modify our set of observations $Obs(bb_i, \phi)$ so that all accesses to $@_A$ in $EETP(bb_i)$ follow the exact $\overline{ATP}(@_A)$ distribution. In fact, enforcing the accesses $@_A$ in $EETP^+(bb_i)$ to globally follow $\overline{ATP}(@_A)$ would require discriminating between hit and miss accesses at $@_A$ and selectively compensating the effects of cache hits on a subset of our $Obs^+(bb_i)$. What can be done, instead, is to enforce our set of observations to follow a *safe over-approximation* of $\overline{ATP}(@_A)$. To this end, we introduce the concept of *padding probability*.

Definition (Padding probability)**.** *A padding probability for $@_A$ inside a path ϕ describes the probability of applying a padding $L_{pad} = L_{miss} - L_{hit}$ to compensate for potentially unobserved cache misses.*

$$PPAD(@_A, \phi) = \left\langle \begin{array}{cc} 0 & L_{pad} \\ 1 - P_{pad}(@_A, \phi) & P_{pad}(@_A, \phi) \end{array} \right\rangle$$

An augmented ATP^+ for $@_A$, independent from ϕ and inclusive of the probabilistic padding is then computed by convolving ($\otimes$ operator) ATP and $PPAD$ as follows:

$$ATP^+(@_A) = ATP(@_A, \phi) \otimes PPAD(@_A, \phi)$$
$$= \left\langle \begin{array}{cc} L_{hit} & L_{miss} \\ P_{hit}(@_A, \phi) & P_{miss}(@_A, \phi) \end{array} \right\rangle \otimes \left\langle \begin{array}{cc} 0 & L_{pad} \\ 1 - P_{pad}(@_A, \phi) & P_{pad}(@_A, \phi) \end{array} \right\rangle$$

The above convolution yields an $ATP^+(@_A)$ with three possible latencies:

1) L_{hit} when $@_A$ is a hit and no padding is applied;
2) L_{miss} when $@_A$ is miss and no padding is applied or $@_A$ is a hit and L_{pad} is applied, since $L_{hit} + L_{pad} = L_{hit} + L_{miss} - L_{hit} = L_{miss}$;
3) $L_{miss} + L_{pad}$ when $@_A$ is miss and padding is applied.

Each latency can happen with a given probability:

1) $P^+_{hit}(@_A) = P_{hit}(@_A, \phi) \cdot (1 - P_{pad}(@_A, \phi))$;
2) $P^+_{miss}(@_A) = P_{miss}(@_A, \phi) \cdot (1 - P_{pad}(@_A, \phi)) + P_{hit}(@_A, \phi) \cdot P_{pad}(@_A, \phi)$;
3) $P^+_{miss+pad}(@_A) = P_{miss}(@_A, \phi) \cdot P_{pad}(@_A, \phi)$

At this point, probabilities do not refer anymore to a specific path ϕ since the application of the probabilistic padding makes them path-independent. Hence, we can rewrite $ATP^+(@_A)$ as follows:

$$ATP^+(@_A) = \left\langle \begin{array}{ccc} L_{hit} & L_{miss} & L_{miss} + L_{pad} \\ P^+_{hit}(@_A) & P^+_{miss}(@_A) & P^+_{miss+pad}(@_A) \end{array} \right\rangle$$

Returning to basic block observations (i.e., elements in EETPs), all memory accesses within any augmented observations $Obs^+(bb_i)$ incur one of the above penalties. In particular, for each access $@_A$ in bb_i we add an L_{pad} padding with a probability $P_{pad}(@_A, \phi)$. All that is required now is to characterize the padding probability $P_{pad}(@_A, \phi)$.

B. Computing the padding probability

We observe that the only requirement on $P_{pad}(@_A, \phi)$ is that $ATP^+(@_A)$ in $Obs^+(bb_i)$ must be path-independent, that is, $ATP^+(@_A)$ must upper-bound $ATP(@_A, \phi)$ for any possible path ϕ in the program that includes *bb*. In other words, our "padded" $ATP^+(@_A)$ can be considered path-independent only if $ATP^+(@_A) \succeq \overline{ATP}(@_A)$, the ATP that upper-bounds all $ATP(@_A, \phi)$.

A graphical representation of the sought relationship between $ATP(@_A, \phi)$, $\overline{ATP}(@_A)$ and $ATP^+(@_A)$, is shown in Figure 3. The relationship between their respective probabilities at execution time L_{hit} and L_{miss} can be expressed by inequalities ⓐ and ⓑ formalized below.

ⓐ $P^+_{hit}(@_A) \leq \overline{P_{hit}}(@_A)$ which translates into
$P_{hit}(@_A, \phi) \cdot (1 - P_{pad}(@_A, \phi)) \leq \overline{P_{hit}}(@_A)$ resulting in:

$$\boxed{P_{pad}(@_A, \phi) \geq 1 - \frac{\overline{P_{hit}}(@_A)}{P_{hit}(@_A, \phi)}}$$

This represents a lower bound to $P_{pad}(@_A, \phi)$ that enables us to enforce path-independence by adding a padding for each access according to a probability distribution.

Similarly for ⓑ we have:

ⓑ $P^+_{hit}(@_A) + P^+_{miss}(@_A) \leq \overline{P_{hit}}(@_A) + \overline{P_{miss}}(@_A)$
which translates into

$$\begin{array}{l} P_{hit}(@_A, \phi) \cdot (1 - P_{pad}(@_A, \phi)) \quad + \\ P_{miss}(@_A, \phi) \cdot (1 - P_{pad}(@_A, \phi)) \quad + \\ \qquad P_{pad}(@_A, \phi) \cdot P_{hit}(@_A, \phi) \leq 1 \end{array}$$

This, instead, results in a tautology (left member in the inequality is always ≤ 1) and adds no information.

Returning to ⓐ, the computation of $P_{pad}(@_A, \phi)$ requires computing $\overline{P_{hit}}(@_A)$, a lower bound to the hit probability of $@_A$ along any possible path, and $P_{hit}(@_A, \phi)$, the exact probability of $@_A$ to be a hit in path ϕ. We compute $\overline{P_{hit}}(@_A)$ using its complement $\overline{P_{miss}}(@_A)$, the upper bound

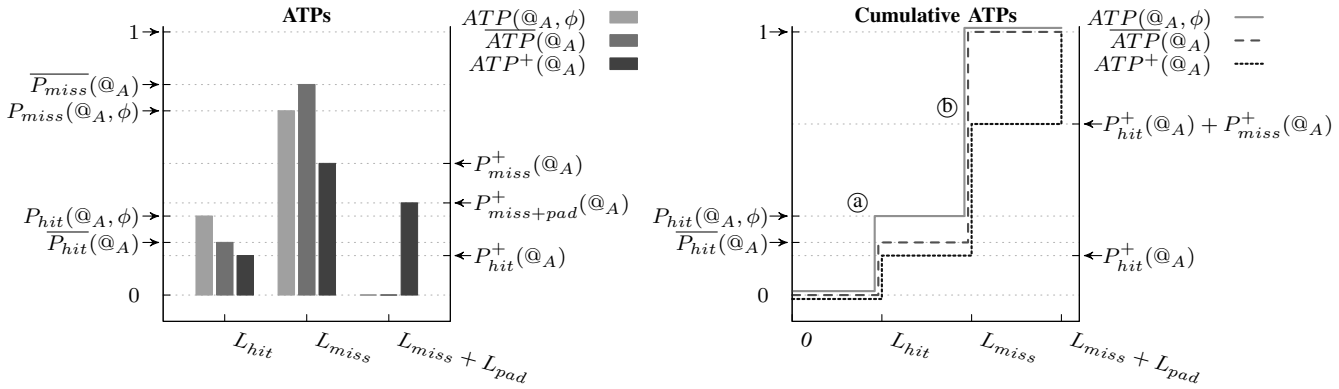


Fig. 3: Graphic representation of $ATP(@_A,\phi)$, $\overline{ATP}(@_A)$, $ATP^+(@_A)$ and the respective cumulative distributions.

to the miss probability for $@_A$ along any possible path: in particular, $\overline{P_{miss}}(@_A)$ will be determined starting from an over-approximation of $P_{miss}(@_A,\phi)$ over any possible path. The exact probability $P_{hit}(@_A,\phi)$ can hardly be computed: an upper bound $uP_{hit}(@_A,\phi)$ can be safely used instead. In fact, using an upper bound to $P_{hit}(@_A,\phi)$ in ⓐ increases the probability of applying the probabilistic padding.

In the computation of the above probabilities, we need upper bounds to the hit and miss probabilities for each access $@_A$ within a given path ϕ. Recall that this per-path probability variation is due to whether the intermediate access between accesses to $@_A$ are hits or misses, which also have a probabilistic nature. In the following we show how these probabilities can be effectively computed by exploiting the concepts of reuse distance and unique accesses. Again, the fact that we do not need to compute the exact hit and miss probabilities is a key point for efficiency.

The *reuse distance* of $@_A$ on a path ϕ is defined as the number of memory blocks mapped to the same set of $@_A$ accessed between $@_A$ and the previous access to the memory block containing $@_A$. If $@_A$'s memory block is never accessed before then the reuse distance is infinite. With *unique accesses*, instead, we refer to the number of *distinct* memory blocks mapped to the same set of $@_A$ accessed in between $@_A$ and the previous access to the memory block containing $@_A$ on a path ϕ. If $@_A$'s memory block is never accessed before then the number of unique accesses is infinite.

By definition, the computation of reuse distance and unique accesses for an access $@_A$ along a given path ϕ requires taking into account exactly all the memory accesses preceding $@_A$ in ϕ. We observe, however, that this computation is not strictly necessary as we can, for example, limit ourselves to consider only the current basic block (performing the access $@_A$) and its immediate predecessor in ϕ. In fact, limiting the computation to this restricted scope is a conservative restriction, as not considering the whole path can only incur pessimism in reuse distance and unique accesses. This relatively restricted scope greatly improves the computational efficiency with only marginal overestimation, as proved by our experiments.

B.1. Computation of $uP_{hit}(@_A,\phi)$

Equation 1 below is used in [21] to compute an upper bound to the probability of miss of an access $@_A$, where w is the number of ways in the cache and $rd(@_A,\phi)$ is the *reuse distance* of $@_A$ in ϕ.

$$uP_{miss}(@_A,\phi)=\begin{cases}1-\left(\frac{w-1}{w}\right)^{rd(@_A,\phi)} & \text{if } rd(@_A,\phi)<w\\ 1 & \text{otherwise}\end{cases} \quad (1)$$

An upper bound to the hit probability of $@_A$ can be defined symmetrically (see Equation 2) by using the number of unique accesses between $@_A$ and the preceding access to the same memory block.

$$uP_{hit}(@_A,\phi)=\begin{cases}1 & \text{if } un(@_A,\phi)<w\\ \left(\frac{w-1}{w}\right)^{un(@_A,\phi)-w+1} & \text{otherwise}\end{cases} \quad (2)$$

If the number of unique accesses $un(@_A,\phi)$ is less than the number of ways w then, in the best case, all accesses are hits and $@_A$ is not evicted (the upper-bound to the hit probability is 1). On the other hand, if $un(@_A,\phi)\geq w$ then at least $un(@_A,\phi)-w+1$ accesses are a miss (they do not fit in cache). In this case, the probability of $@_A$ to be a hit equals, at most, the probability of $un(@_A)-w+1$ accesses to evict some other cache lines. Both $rd(@_A,\phi)$ and $un(@_A,\phi)$ can be efficiently computed on a restricted scope, considering only one predecessor basic block.

B.2. Computation of $\overline{P_{hit}}(@_A)$

$\overline{P_{hit}}(@_A)$ is defined $1-\overline{P_{miss}}(@_A)$. $\overline{P_{miss}}(@_A)$ can be computed as an upper-bound to the exact miss probability $P_{miss}(@_A,\phi)$ for any possible path $\phi\in\mathcal{P}$. Again, since the exact probability is hard to compute, we use $uP_{miss}(@_A,\phi)$ from Equation 1 as a safe upper-bound instead:

$$\overline{P_{miss}}(@_A)\geq\max_{\phi}\{uP_{miss}(@_A,\phi)\}\geq\max_{\phi}\{P_{miss}(@_A,\phi)\}$$

In principle, the computation of $\overline{P_{miss}}(@_A)$ for access $@_A$ within a basic block bb_i would require to compute $uP_{miss}(@_A,\phi)$ for all ϕ that include bb_i, which is a complex and onerous computation. For determining the reuse distance, this computation is not strictly necessary, as we can limit it to the set of basic blocks that immediately precede bb_i. This optimization prunes the inherent complexity of the control flow graph of the program and makes the computation of $\overline{P_{miss}}(@_A)$ computationally efficient.

We use Algorithm 1 to compute $\overline{P_{miss}}(@_A)$ for an access $@_A$ in a basic block bb_i. As anticipated, the algorithm scope is limited to the set of basic blocks $\{bb_j,\ldots,bb_k\}$ that can be executed just before bb_i (its immediate predecessors). For each basic block bb_l in $\{bb_j,\ldots,bb_k\}$ the reuse distance $rd(@_A,@(bb_l\rightarrow bb_i))$ is computed. $@(bb_l\rightarrow bb_i)$ is the sequence of accesses in bb_l followed by accesses in bb_i. If any $rd(@_A,@(bb_l\rightarrow bb_i))$ is infinite (i.e. $@_A$ is not accessed before in $@(bb_l\rightarrow bb_i)$) then we assume that $@_A$ is always a miss in the worst case, that is, $\overline{P_{miss}}(@_A)=1$. Otherwise we call $\overline{rd}(@_A)$ the maximum reuse distance computed with a predecessor selected from $\{bb_j,\ldots,bb_k\}$ and define $\overline{P_{miss}}(@_A)$

and $\overline{P_{hit}}(@_A)$ as:

$$\overline{P_{miss}}(@_A) = \begin{cases} 1 - \left(\frac{w-1}{w}\right)^{\overline{rd}(@_A)} & \text{if } \overline{rd}(@_A) < w \\ 1 & \text{otherwise} \end{cases}$$

$$\overline{P_{hit}}(@_A) = 1 - \overline{P_{miss}}(@_A)$$

Algorithm 1 Computation of $\overline{P_{miss}}(@_A)$, an upper-bound to any possible miss probability for $@_A$

```
w := number of cache ways
@_A := cache access in a basic block bb_i
rd(@_A) := 0
{bb_j, ..., bb_k} := predecessors of bb_i in P
for all bb_l ∈ {bb_j, ..., bb_k} do
    @(bb_l bb_i) := accesses performed by bb_l and bb_i
    rd(@_A, @(bb_l bb_i)) := reuse distance for @_A
    if (rd(@_A, @(bb_l bb_i)) is ∞) then
        return 1
    else if (rd(@_A, @(bb_l bb_i)) > rd(@_A)) then
        rd(@_A) := rd(@_A, @(bb_l bb_i))
    end if
end for
if rd(@_A) ≥ w then
    return 1
else
    return 1 − ((w−1)/w)^rd(@_A)
end if
```

By construction, $\overline{P_{miss}}(@_A)$ is an upper bound to any possible miss probability for $@_A$ through any path; similarly, $\overline{P_{hit}}(@_A)$ is a lower bound to any possible hit probability for $@_A$ through any path.

In Algorithm 1 we use very little information on the basic blocks that may precede bb_i. As our results show, the computed bounds are reasonably tight in spite of this approximation. Increasing the analysis scope and considering predecessors of predecessors (up to the whole path) could tighten the computation of $\overline{ATP}(@_A)$, but at higher computational costs.

B.3. Computing a value for P_{pad}

Once means to compute $\overline{P_{hit}}(@_A)$ and $uP_{hit}(@_A, \phi)$ are provided, the first inequality ⓐ can be used to compute a sound value for $P_{pad}(@_A, \phi)$, as shown in Equation 3.

$$P_{pad}(@_A, \phi) = \begin{cases} 0 & \text{if } uP_{hit}(@_A, \phi) = 0 \\ 1 - \dfrac{\overline{P_{hit}}(@_A)}{uP_{hit}(@_A, \phi)} & \text{otherwise} \end{cases} \quad (3)$$

Note that $\overline{P_{hit}}(@_A)$ and $uP_{hit}(@_A, \phi)$ are valid probabilities ($\in [0, 1]$) and $\overline{P_{hit}}(@_A) \leq uP_{hit}(@_A, \phi)$ by construction, as $\overline{P_{hit}}(@_A)$ is a lower bound to any possible hit probability for $@_A$. Therefore $P_{pad}(@_A, \phi)$ is a valid probability. Moreover, from Equation 3 we can draw the following conclusions on the behaviour of $P_{pad}(@_A, \phi)$:

- $P_{pad}(@_A, \phi)$ decreases as $\overline{P_{hit}}(@_A)$ grows: the less likely $@_A$ will be a miss in the worst case the lower the need to apply a padding;
- $P_{pad}(@_A, \phi)$ decreases as $uP_{hit}(@_A, \phi)$ decreases: the more likely $@_A$ is a miss in the measured path the less needed is a padding to negatively compensate for the worst case.

Intuitively these observations confirm that Equation 3 provides a probability for the application of a padding to compensate for unobserved misses. An explicative example on how probabilistic padding is applied to the observations over a basic block is provided in Annex I.

V. Extending path coverage

So far we have discussed a probabilistic approach to make the observed execution times $Obs(bb_i)$ of each basic block *path-independent* by systematically adding a probabilistic padding. After applying this padding to each observation that forms its EETP, each basic block bb_i is characterized by an augmented $\text{EETP}^+(bb_i)$. Since $Obs(bb_i)$ in $\text{EETP}(bb_i)$ have been padded in $\text{EETP}^+(bb_i)$, $\text{EETP}^+(bb_i)$ dominates $\text{EETP}(bb_i)$ in the sense that each $Obs^+(bb_i)$ in $\text{EETP}^+(bb_i)$ is higher or equal to its counterpart in $\text{EETP}(bb_i)$. Our next step is to reap the path-independence property (relative to single basic blocks) at the level of end-to-end observations, to improve the representativeness of the observed paths. The intuition here is that we first break the relationship between observations and their respective path, to be later able to reconstruct valid upper bounds for any path in the program.

Given a program $\mathcal{P}$, we consider the set of end-to-end paths in $\mathcal{P}$ as: $\Phi(\mathcal{P}) = \{\phi_i | \phi_i \textit{ is an end-to-end path in } \mathcal{P}\}$. $\Phi(\mathcal{P})$ is potentially an infinite set due to typical constructs in a program, such as loops and recursion. However, we can focus on $\Phi(\overline{\mathcal{P}})$, a finite subset of $\Phi(\mathcal{P})$, as determined by the common restrictions in timing analysis, such as known loop bounds and absence of recursion. Consequently, also the set of end-to-end observations collected at analysis time are relative to some of the paths in $\Phi(\overline{\mathcal{P}})$, that is, $\Phi_{obs}(\overline{\mathcal{P}}) \subset \Phi(\overline{\mathcal{P}})$.

Much like $\Phi(\mathcal{P})$, $\Phi(\overline{\mathcal{P}})$ may also include infeasible paths, that is, paths that are structurally possible, but can never happen in practice due to functional constraints (e.g., mutually exclusive conditions over two branches). Known techniques [1] can be used to reduce the risk of pessimism incurred considering infeasible paths. Our objective is to augment the representativeness of the observations made over $\Phi_{obs}(\overline{\mathcal{P}})$ up to include all potentially unobserved paths in $\Phi(\overline{\mathcal{P}})$. In practice, we aim at deriving safe over-approximations of $\text{EETP}(\phi_i)$ for all paths in $\Phi(\overline{\mathcal{P}})$ that have not been actually observed.

In the following we present the rationale for our approach and describe a practical procedure for extending the representativeness of the collected observations. We introduce the concepts of path construction and describe how they can be used to compute safe over-approximations of multiple paths.

We first recall two relevant properties:

Property 1. *In the absence of timing anomalies [22], the worst-case execution time of a program path is determined by accumulation of the worst-case execution times of its constituent basic blocks (i.e., local worst leads to global worst).*

Property 2. *Under the path-independence condition, any execution time value in $\text{EETP}^+(bb_i)$ is an equally valid execution time upper bound for bb_i, regardless of the traversed path.*

Under these conditions, we can sequentially combine augmented basic blocks profiles EETP^+ to artificially build an EETP for all unobserved paths in $\Phi(\overline{\mathcal{P}})$. Our approach to execution time construction builds an artificial observation $\widehat{Obs}(\phi_i)$ for all paths ϕ_i that have not been stressed by summing up the execution times of each basic block along the path (we can equally proceed bottom up or top down). For each basic block bb_i, an execution time in $\text{EETP}^+(bb_i)$ is selected simply by applying a random sampling over the set of $Obs^+(bb_i)$ (actually a multiset) so that frequencies in $\text{EETP}^+(bb_i)$ are kept. Several synthetic observations for each path need to be constructed. This ensures that $\widehat{\text{EETP}}(\phi)$ exhibits, if not the same empirical distribution as $\text{EETP}(\phi)$,

at least a distribution that leads to a higher pWCET for any exceedance probability. The procedure is more formally described in Algorithm 2.

Algorithm 2 Exhaustive enumeration based on SRS

```
1:  for all unobserved φ_k ∈ Φ(P̄) do
2:      ÊETP(φ_k) ← nihil
3:      repeat
4:          Ôbs(φ_k) ← 0
5:          for all bb_i ∈ φ_k do
6:              Ôbs(φ_k) += SRS(EETP+(bb_i))
7:          end for
8:          ÊETP(φ_k) = ÊETP(φ_k) ∪ Ôbs(φ_k)
9:      until MBPTA convergence criteria is met
10: end for
```

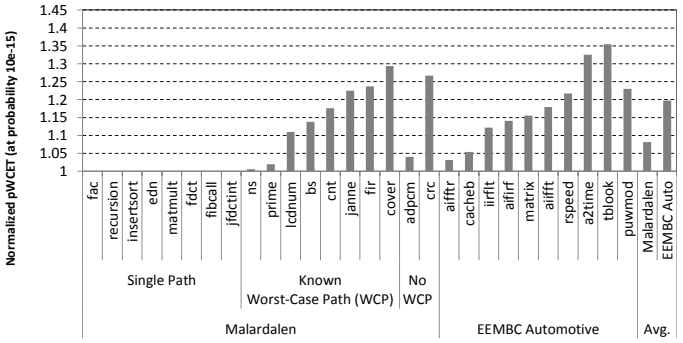
For each path ϕ_k in $\Phi(\overline{\mathcal{P}})$ that has never been traversed we collect a number of artificial execution times by iterating over all the basic blocks in the path. An illustrative example is provided in Appendix I. In Algorithm 2 (line 6) we use a temporary variable $\widehat{Obs}(\phi_k)$ to accumulate the cost of each basic block along the path. Each block bb_i contributes to that variable with a single value selected from the augmented profile $\mathrm{EETP}^{+}(bb_i)$. The simple random sampling[6] (SRS) method can be used as, thanks to path independence, any value in $\mathrm{EETP}^{+}(bb_i)$ can be safely used. The end-to-end artificial observation contributes to the artificial execution time profile for the path under analysis (line 8). The set union operator $\cup$ is used here as $\widehat{Obs}(\phi_k)$ contributes to the multiset of observations from which $\widehat{\mathrm{EETP}}(\phi_k)$ is derived. The outermost loop in Algorithm 2 (lines 3-9) guarantees that the generation of a synthetic observation is repeated until we collect a sufficient number of elements in $\widehat{\mathrm{EETP}}(\phi_k)$. A moderately large number of observations (a few hundreds) is typically required in the first place. Some empirical considerations in this regard are provided in Section VI.

The $\widehat{\mathrm{EETP}}(\phi_k)$ for all $\phi_k \in \Phi(\overline{\mathcal{P}})$ will be finally merged together to form a global EETP for $\mathcal{P}$, on which MBPTA can be applied. EVT warrants that the execution times of the longest path dominates all other paths [4]. As noted in Section III, the EPC process is applied to different memory placements so that EVT can thoroughly capture random placement as a source of variability.

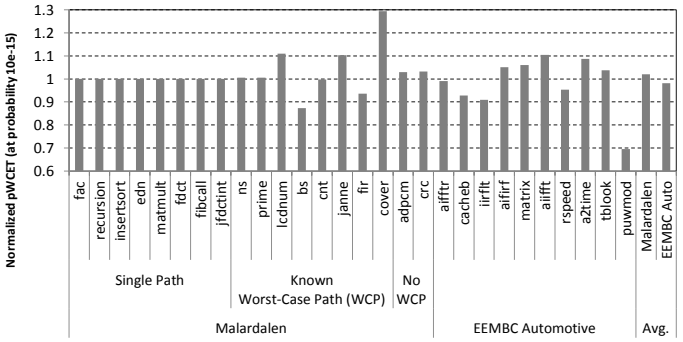
VI. Evaluation

Our experiments had multiple objectives. First, we wanted to assess EPC against original MBPTA, knowing that EPC provides more trustworthy results than plain MBPTA. Second, we aimed at comparing EPC with PUB in terms of tightness, as they both provide trustworthy results. An evaluation of EPC against STA approaches is out of the scope of this paper: an indirect comparison can be drawn from [23], which contrasts STA to MBPTA. Finally, we were also interested in evaluating the computational cost of applying EPC. In our evaluation we used benchmarks from the Mälardalen [24] and the EEMBC Autobench [25] suites. We run our experiments on a cycle-level platform simulator based on SoCLib [26], where timing anomalies are prevented by construction. At core level, we model a 5-stage in-order processor, where each instruction incurs a fixed latency, and core hardware units are fully pipelined. This rules out any core-level dependence between subsequent instructions, hence among different basic blocks.

[6]With SRS all members of the population have the same chance of being selected.



(a) Values normalized to MBPTA outputs.



(b) Values normalized to PUB outputs.

Fig. 4: pWCET estimates provided by EPC at an exceedance probability of 10^{-15}.

Core-level basic block latencies are thus path independent by design and require no EPC support. For caches, we model first-level instruction and data caches (I-cache and D-cache). Both I- and (write-through) D-cache are 4-KB set-associative caches with 4 ways and 32 bytes lines implementing random placement and replacement [13]. The latency is set to 1 clock cycle for a hit and 10 clock cycles for a miss for both caches. EPC is configured to only consider direct predecessors of the basic block under study when computing probabilistic padding. Better (tighter) results could be achieved, at the cost of more complexity, by considering more than one predecessor.

EPC vs MBPTA. MBPTA, in its original formulation, is known to provide trustworthy results only for the timing phenomena observed in the program paths traversed in the measurement runs. EPC synthetic path coverage produces fully trustworthy results that are valid for any path in a program. We aimed at evaluating the increase in pWCET values incurred by EPC, being aware that such increase stems in part from the additional paths considered in the analysis and in part from the overestimation introduced by applying a probabilistic padding. Figure 4(a) summarizes the results of the comparison of EPC and MBPTA. The evaluation on single-path benchmarks, the leftmost group in Figure 4(a), shows no difference between EPC and MBPTA, as EPC simply falls back to the latter.

For some of the multi-path programs in the Mälardalen benchmarks (`ns`, `prime`, `lcdnum`, `bs`, `cnt`, `janne`, `fir` and `cover`), the worst-case path is known and can thus be covered by measurement runs. MBPTA can then compute a pWCET that is valid for all paths in the program just by collecting measurements over the worst-case path. Therefore, only on these particular benchmarks, the difference in pWCET results between EPC and MBPTA is completely ascribable to the overestimation incurred by EPC. As shown in Figure 4(a)

the increase in the pWCET is always below 23.6% (13% on average) except for `cover`, for which EPC overestimates the pWCET by nearly 30%. The rise in pessimism in `cover` is caused by the presence of loops that include a large `switch` statement (with up to 120 `cases`). At each iteration in those loops, in the absence of further guidance, EPC systematically applies a conservative probabilistic padding to compensate for the effects of the I-cache. For the Mälardalen `adpcm` and `crc` and all EEMBC's, MBPTA does not provide pWCET results that are valid for all paths. Hence, the average 19% pWCET increase that EPC has over MBPTA cannot be interpreted as an overestimation, as there might be a path not covered by the input vectors fed into MBPTA whose traversal would increase the MBPTA pWCET estimates.

EPC vs PUB. Unlike MBPTA and similarly to PUB [5], our approach provides fully trustworthy results that are valid regardless of the paths observed at analysis time. Under that premise, we were interested in whether EPC could produce tighter results (i.e., lower pWCET values) than PUB. The results of our comparison are summarized in Figure 4(b). Again, the leftmost block of benchmarks comprises only single-path programs where EPC and PUB do not add to standard MBPTA.

The comparison between EPC and PUB shows that in the average case these methods produce comparable results in terms of tightness. In fact, for the benchmarks for which MBPTA cannot determine the pWCET for all paths, the EPC results are on average 1% lower than those obtained with PUB. More in general, considering all multi-path benchmarks from both suites, EPC produces pWCET values that exceed those computed with PUB by only 0.07% on average.

When comparing EPC and MBPTA, we already observed how EPC may introduce additional overestimation owing to certain code construct (e.g., loop with large branch factor as in `cover`). As expected, in those cases PUB offers tighter results. However, for some other benchmarks (EEMBC's `aifftr`, `cacheb`, `iirflt`, `rspeed`, `puwmod` and Mälardalen's `bs`, `cnt` and `fir`), EPC computes tighter bounds than PUB. The differences between EPC in PUB results are explained by the fact that EPC adds a probabilistic padding to over-approximate, as tightly as possible, the execution time distribution of the worst-case path. PUB instead forces all paths to have a similar execution time distribution by adding instructions and memory accesses to the program that do not affect the semantics but balance the timing behaviour of branches. On some benchmarks, such instructions may cause the program to exhibit either (i) higher execution times than the original worst case or, (ii) higher variability of execution times. Both cases cause PUB to compute a pWCET higher than EPC for low exceedance probabilities. Focusing on EEMBC, whose programs have higher complexity and longer execution times than Mälardalen's, we see that the EPC results are 2% tighter than those computed with PUB. This result shows that, by using a modest amount of information on the program structure, EPC performs as good as PUB and solves the path coverage problem for MBPTA.

Overall, in terms of tightness in the pWCET estimates EPC is competitive w.r.t. PUB. Further, unlike PUB, the strong argument in favour of EPC is that it does not require changes to the program code. Although only effective at analysis time, those changes entail additional (and unwanted) costs in the certification and validation process as well as the procurement of qualified technology. EPC, instead, only requires that precise bounds to the latency of certain hardware events, such as cache misses, are known for the platform under consideration.

Computation requirements. Owing to its iterative nature, the EPC process may require considerable computation time to derive pWCET estimates. The padding computation as well as the execution-time construction concur to the overall complexity. The former phase is applied to different measurements for several cache placements and can therefore be parallelized. The latter phase is currently implemented as a single-threaded process, though it could certainly use some parallelism. All experiments were run on a standard mid-end laptop. 500 different placements were considered and 500 measurements collected for each random placement. Our results for Mälardalen and EEMBC Autobench benchmarks – excluding single-path benchmarks, of no interest for EPC – show that the padding computation took less than 100 minutes on average (300 in the longest case) when running in parallel on 8 cores. Path construction (in its single-thread implementation) took 63 minutes on average (less than 183 minutes in the longest run). Note that our prototype is currently implemented as a mixture of Perl and Python scripts, which are not the best choice for performance. EPC is arguably able to deliver results with affordable time costs even when run on plain commodity processors.

VII. Related Work

Massive research effort has been expended in the last decades to address the challenge of timing analysis on modern hardware platforms [1]. Probabilistic timing analysis approaches have recently gained interest as a means to mitigate the effects of history-dependent jitter by exploiting deterministic *and* time-randomized hardware resources such as, for example, randomized caches [13]. Although the use of randomized caches might lead to some complex timing effects, as noted in [16], appropriate solutions to detect and overcome those phenomena do exist [17], [18]. In this paper we focus on the measurement-based variant of probabilistic timing analysis [4], [9] and address the problem, typical of measurement-based methods, of path coverage as it affect representativeness (and trustworthiness) of the obtained results. Several techniques address the path-coverage problem and the cost and complexity of catering for the desired degree of coverage.

The Single-Path Approach [27] reduces complex programs to a single execution path on processors that provide constant-time predicated instructions. The Single-Path Approach needs compiler support to leverage hardware predication. With EPC, we do away with any modification to the compiler.

Automatic approaches to generate input vectors for controlled path coverage have been proposed, for example, in [28], [29]. These methods typically suggest to use genetic algorithms in combination with model checking to heuristically increase the trustworthiness of their measurements. None of the automatic input data generation techniques, however, has been proven to always generate inputs that guarantee full path coverage or that capture the worst-case path. Our technique, when applied on time-randomized hardware, compensate for the limited coverage of user-defined input vectors, without relying on heuristically-generated inputs.

Hybrid techniques that integrate static timing analysis and measurements have been proposed, e.g. [30], [31], to combine static information on the program and measurements thus enabling the user to trade off between precision and cost. While hybrid techniques allow balancing between precision and costs,

they do not warrant trustworthiness. Like hybrid techniques, our approach divides the control flow graph of a program into basic blocks. EPC, however, is combined with MBPTA, which provides confidence in the obtained results, by ensuring that the computed pWCET is exceeded only with a probability that is lower than a user-provided threshold.

Finally, PUB [5] is the approach that presents the most similarity with our work. PUB artificially balances the different branches of conditional control flow constructs by inserting core and cache access instructions so that each branch is a safe upper-bound for all the alternative branches. Execution time measurements are taken on the extended version of the program, which is used at analysis time only. By construction, the derived pWCET is also valid for the original program. In contrast with our approach, PUB requires a qualified compiler to generate a semantically-preserving extended version of the program. EPC instead operates directly on the collected measurements and thus does not require any program modification.

Stochastic approaches [32]–[34] have been also proposed to study the probability distribution of the response times of each task in a system to model the probability of missing a deadline. These approaches, however, only assume tasks to exhibit probabilistic timing behaviour and do not study the determination of pWCET estimates.

VIII. Conclusions

MBPTA provides trustworthy pWCET bounds for multi-path programs as long as all timing phenomena of interest have been observed in the paths traversed in the measurement runs. Achieving full path coverage or identifying the worst-case path by means of observations, however, is not feasible in general. In this paper we presented a new approach, EPC, that extends MBPTA's confidence in the computed bounds, even for unobserved paths, and requires only basic block coverage. The execution times of each basic block are made path-independent by applying a probabilistic penalty that mitigates the benefits that might have been accrued by specific path traversals. Path-independent execution times are then combined to construct synthetic end-to-end execution times for unobserved paths. Experimental results show that EPC incurs a competitive amount of overestimation in comparison with both standard MBPTA and PUB. In contrast with PUB, EPC does not require changing the program code. As future work, we plan to further investigate whether collapsing complex conditional constructs to a single EETP could improve the trade off between the complexity of the analysis and the tightness of the results.

Acknowledgements

This work has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 611085 (PROXIMA [12]). This work and has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, the HiPEAC Network of Excellence and COST Action IC1202: Timing Analysis On Code-Level (TACLe). 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] R. Wilhelm *et al.*, "The worst-case execution time problem: overview of methods and survey of tools," *Trans. on Embedded Computing Systems*, vol. 7, no. 3, pp. 1–53, 2008.

[2] J. Abella *et al.*, "WCET analysis methods: Pitfalls and challenges on their trustworthiness," in *SIES*, 2015.

[3] Special Committee of RTCA, "DO-178C, Software Considerations in Airborne Systems and Equipment Certification," 2011.

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

[5] L. Kosmidis *et al.*, "PUB: Path upper-bounding for measurement-based probabilistic timing analysis," in *ECRTS*, 2014.

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

[7] S. Kotz and S. Nadarajah, *Extreme Value Distributions: Theory and Applications*. Imperial College Press, 2000.

[8] L. David and I. Puaut, "Static determination of probabilistic execution times," in *ECRTS*, 2004.

[9] F. Cazorla *et al.*, "PROARTIS: Probabilistically analysable real-time systems," *Transactions on Embedded Computing Systems*, 2013.

[10] L. Kosmidis *et al.*, "Probabilistic timing analysis and its impact on processor architecture," in *Euromicro DSD*, 2014.

[11] M. Paolieri *et al.*, "Hardware support for wcet analysis of hard real-time multicore systems," in *ISCA*, 2009.

[12] "Probabilistic real-time control of mixed-criticality multicore and manycore systems (PROXIMA)." [Online]. Available: http://www.proxima-project.eu/

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

[14] Aeroflex Gaisler, *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and User's Manual*, 2011. [Online]. Available: http://microelectronics.esa.int/ngmp/LEON4-NGMP-DRAFT-1-6.pdf

[15] F. Cazorla *et al.*, "Upper-bounding program execution time with extreme value theory," in *WCET Analysis Workshop*, 2013.

[16] J. Reineke, "Randomized caches considered harmful in hard real-time systems," *LITES*, vol. 1, no. 1, pp. 03:1–03:13, 2014. [Online]. Available: http://dx.doi.org/10.4230/LITES-v001-i001-a003

[17] J. Abella *et al.*, "Heart of Gold: Making the improbable happen to extend coverage in probabilistic timing analysis," in *ECRTS*, 2014.

[18] E. Mezzetti *et al.*, "Randomized caches can be pretty useful to hard real-time systems," *LITES*, vol. 2, no. 1, pp. 01:1–01:10, 2015. [Online]. Available: http://dx.doi.org/10.4230/LITES-v002-i001-a001

[19] "Nexus 5001 forum." [Online]. Available: http://www.nexus5001.org

[20] "Rapita Systems Ltd., Rapita Verification Suite (RVS)." [Online]. Available: http://www.rapitasystems.com/products/rvs

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

[22] T. Lundqvist and P. Stenström, "Timing anomalies in dynamically scheduled microprocessors," in *RTSS*, 1999.

[23] J. Abella *et al.*, "On the comparison of deterministic and probabilistic wcet estimation techniques," in *ECRTS*, 2014.

[24] J. Gustafsson *et al.*, "The Mälardalen WCET benchmarks-past, present and future," in *WCET Workshop*, 2010.

[25] J. A. Poovey *et al.*, "A benchmark characterization of the eembc benchmark suite," *IEEE Micro*, vol. 29, no. 5, pp. 18–29, 2009.

[26] "SoCLib simulation environment." [Online]. Available: http://www.soclib.fr/trac/dev

[27] P. Puschner, "The single-path approach towards WCET-analysable software," in *International Conference on Industrial Technology*, 2003.

[28] S. Bünte *et al.*, "Improving the confidence in measurement-based timing analysis," in *ISORC*, 2011.

[29] I. Wenzel *et al.*, "Automatic timing model generation by cfg partitioning and model checking," in *DATE*, 2005.

[30] M. Zolda, S. Bünte, and R. Kirner, "Towards adaptable control flow segmentation for measurement-based execution time analysis," in *RTNS*, 2009.

[31] M. Zolda, S. Bünte, and R. Kirner, "Context-sensitive measurement-based worst-case execution time estimation," in *RTCSA*, 2011.

[32] T.-S. Tia *et al.*, "Probabilistic performance guarantee for real-time tasks with varying computation times," in *RTAS*, 1995.

[33] L. Palopoli *et al.*, "An analytical bound for probabilistic deadlines," in *ECRTS*, 2012.

[34] J. Diaz *et al.*, "Stochastic analysis of periodic real-time systems," in *RTSS*, 2002.

APPENDIX I
AN ILLUSTRATIVE EXAMPLE OF APPLICATION OF EPC

We provide here an example of application of EPC on a simple fragment drawn from an arbitrarily complex program. We assume to have selected a cache placement and we illustrate the tree steps of EPC, as introduced in Section III: i) *Execution time collection*; ii) *Probabilistic basic block padding*; and iii) *Synthetic path construction*.

For the sake of simplicity, we limit our example to the application of EPC to Data cache only and show how a probabilistic padding for data accesses can be computed. Analogous reasoning can be applied to instruction accesses. We assume the data cache to be a 4-way set associative time-randomized cache. Latency of a hit and a miss are 1 and 10 clock cycles, respectively.

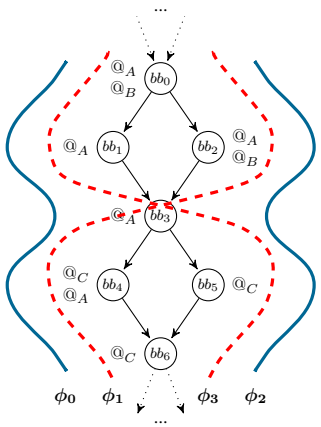


Fig. 5: Example program made of two sequential conditional constructs. Each basic block has associated data accesses ($@_X$). Solid paths (ϕ_0 and ϕ_2) are explicitly executed. Dashed paths (ϕ_1 and ϕ_3) are synthetically covered by EPC.

Figure 5 shows our example, consisting in seven basic blocks, which form two sequential `if-then-else` statements, part of a larger control flow graph. Each basic block performs some data accesses. Accessed addresses are annotated beside each basic block in Figure 5. Under the selected placement we know $@_A$ and $@_B$ to be mapped to the same cache set (e.g. conflict in the cache) while $@_C$ is mapped elsewhere.

A. Execution Time Collection

We consider the sub-graph in Figure 5 to be a part of a more complex program. We assume that measurements were collected for each basic block by traversing paths ϕ_0 and ϕ_2 only; such measurements are used to determine the empirical execution time profiles (EETP) for each of the represented basic blocks. To keep the example simple and effective we assume that all EETP are made of 1000 measurements. In reality, EETPs might have different sizes that depend on how input vectors cover different basic blocks. Moreover, EETP(bb_1) and EETP(bb_4) are composed only by observations collected through path ϕ_0. Analogously, EETP(bb_2) are EETP(bb_5) are composed by observations collected by traversing path ϕ_2. Finally EETP(bb_0), EETP(bb_3) and EETP(bb_6) contain an equal number of measurements collected through path ϕ_0 (i.e. $Obs(bb_3, \phi_0)$) and through path ϕ_2 (i.e. $Obs(bb_3, \phi_2)$). The detailed EETPs are reported in Table I.

EETP	Execution Time	Number of Observations
EETP(bb_0)	20	1000
EETP(bb_1)	1 10	750 250
EETP(bb_2)	2 11 20	750 187 63
EETP(bb_3)	1 10	992 8
EETP(bb_4)	11	1000
EETP(bb_5)	10	1000
EETP(bb_6)	10	1000

TABLE I: Possible empirical execution time profiles for the example program in Figure 5.

B. Probabilistic Basic Block Padding

The computation (and application) of a probabilistic padding is perhaps the most intricate step in EPC. In the following we focus on basic blocks bb_3 and bb_4 as they are more interesting here. We assume to have no information on bb_0 predecessors and simply pretend, for the sake of simplicity, that there is no need to apply any padding.

B.1. Probabilistic padding for bb_3

As previously stated, EETP(bb_3) contains observations obtained on both paths ϕ_0 and ϕ_2. We compute a probabilistic padding for access $@_A$ in bb_3 for both paths, $PPAD(@_A, \phi_0)$ and $PPAD(@_A, \phi_2)$, exploting information only on bb_3's direct predecessors bb_1 and bb_2. We recall that $PPAD(@_A, \phi)$ can be computed using Equation 3, which we report here for convenience.

$$P_{pad}(@_A, \phi) = \begin{cases} 0 & \text{if } uP_{hit}(@_A, \phi) = 0 \\ 1 - \dfrac{\overline{P_{hit}}(@_A)}{uP_{hit}(@_A, \phi)} & \text{otherwise} \end{cases}$$

Therefore we need to compute both $\overline{P_{hit}}(@_A)$ and $uP_{hit}(@_A, \phi)$ for ϕ_0 and ϕ_2.

1) $\overline{P_{hit}}(@_A)$: In the computation of the reuse distance, functional to compute $\overline{P_{hit}}$, we consider only the two immediate predecessors of bb_3 along paths ϕ_o and ϕ_2, that is bb_1 and bb_2. Since address $@_A$ is accessed in both bb_1 and bb_2, its maximum reuse distance is $\overline{rd}(@_A) = 1$. By applying Algorithm 1 (exploiting Equation 1) to compute $\overline{P_{miss}}(@_A)$, we derive:

$$\overline{P_{hit}}(@_A) = 1 - \overline{P_{miss}}(@_A) = 1 - \frac{1}{4} = \frac{3}{4}$$

2) $uP_{hit}(@_A, \phi_0)$: Again, in the computation of the number of unique accesses, we limit ourselves to the current basic block and its immediate predecessor. When bb_3 is reached through bb_1 (path ϕ_0) the number of unique accesses $un(@_A, \phi_0)$ is 0. Therefore, according to Equation 2, $uP_{hit}(@_A, \phi_0) = 1$. This leads to a probability of applying the padding to observations in EETP(bb_3) collected through ϕ_0 defined as follows:

$$P_{pad}(@_A, \phi_0) = 1 - \frac{\frac{3}{4}}{1} = \frac{1}{4}$$

3) $uP_{hit}(@_A, \phi_2)$: Similarly, when bb_3 is reached through bb_2 (path ϕ_2) the number of unique accesses $un(@_A, \phi_2)$ is always 1, which again leads to $uP_{hit}(@_A, \phi_2) = 1$. The probability of applying a padding is $P_{pad}(@_A, \phi_2) = \frac{1}{4}$.

B.2. Probabilistic padding for bb_4

Basic block bb_4 deserves our attention as it performs accesses to two addresses ($@_C$ and $@_A$) that we assume to be mapped to different cache sets.

Focusing on $@_C$, we observe that no access to the same address is performed in bb_4 direct predecessor (bb_3). Owing to our restricted scope in the computation of reuse distance, this means that no information on previous accesses is available; the reuse distance is therefore assumed to be infinite and by definition:

$$\overline{P_{hit}}(@_C) = 0$$

It therefore follows that, according to Equation 3, the probability of applying a padding to bb_4 observations is

$$P_{pad}(@_C, \phi_0) = 1$$

This is in fact one of the unlucky cases in which, due to lack of information on recent history, we apply a probabilistic padding for an access that was already (always) a miss.

With respect to any accesses $@_A$, on the other hand, we observe that they cannot suffer any interference from $@_C$ as they map to different cache sets. Therefore the number of unique accesses occurring between $@_A$ and the previous access to same address (in bb_3) is exactly 0, which implies

$$uP_{hit}(@_A, \phi_0) = 1$$

Similarly, since the same address is accessed in the unique predecessor of bb_4, it has a reuse distance of 0 and thus

$$\overline{P_{hit}}(@_A) = 1$$

Through Equation 3, we obtain a probability of padding

$$P_{pad}(@_A, \phi_0) = 1 - \frac{1}{1} = 0$$

B.3. Applying the probabilistic padding

Once the padding probability is computed for an access $@_X$ the latency of padding ($L_{pad} = L_{miss} - L_{hit} = 10 - 1 = 9$) is accordingly added to the execution times of the corresponding basic block. If we consider bb_3, for instance, all observations that form $\mathrm{EETP}(bb_3)$ are augmented by L_{pad} for the access $@_A$ with a probability $\frac{1}{4}$. For all observations $Obs(bb_3, \phi)$, in fact, we perform the following operation:

$$Obs^+(bb_3) = \begin{cases} Obs(bb_3, \phi) + L_{pad} & \text{if } rand() \leq \frac{1}{4} \\ Obs(bb_3, \phi) & \text{otherwise} \end{cases}$$

Applying probabilistic padding to all basic blocks in the example leads augmented execution time profiles EETP^+ reported in Table II.

C. Synthetic Path Construction

The execution time construction phase is relatively simple. To build synthetic observations for the unobserved paths ϕ_1 and ϕ_3, shown in Figure 5, we random sample the augmented empirical execution time profiles EETP^+ of the basic blocks that constitute those paths. Sampling is performed on the superset of augmented observations Obs^+ that form EETP^+. Note that all values in EETP^+ are path-independent and thus each EETP^+ is a valid over-approximation of the original EETP. Below we provide an example of execution time construction for path ϕ_1 in which, for each basic block, the most frequent value happens to be selected.

EETP	Execution Time	Number of Observations
$\mathrm{EETP}^+(bb_0)$	20	1000
$\mathrm{EETP}^+(bb_1)$	1	562
	10	375
	19	63
$\mathrm{EETP}^+(bb_2)$	2	421
	11	386
	20	153
	29	36
	38	4
$\mathrm{EETP}^+(bb_3)$	1	744
	10	254
	19	2
$\mathrm{EETP}^+(bb_4)$	20	1000
$\mathrm{EETP}^+(bb_5)$	19	1000
$\mathrm{EETP}^+(bb_6)$	10	1000

TABLE II: Augmented EETPs obtained by applying the probabilistic padding to empirical execution time profiles in Table I.

$$\begin{aligned}\widehat{\mathrm{Obs}}(\phi_1) = {} & SRS(\mathrm{EETP}^+(bb_0)) + SRS(\mathrm{EETP}^+(bb_1)) + \\ & SRS(\mathrm{EETP}^+(bb_3)) + SRS(\mathrm{EETP}^+(bb_5)) + \\ & SRS(\mathrm{EETP}^+(bb_6)) \\ = {} & 20 + 1 + 1 + 20 + 10 = 52\end{aligned}$$

Random sampling ensures that also less frequent values for some basic blocks have a chance to be selected:

$$\begin{aligned}\widehat{\mathrm{Obs}}(\phi_1) = {} & SRS(\mathrm{EETP}^+(bb_0)) + SRS(\mathrm{EETP}^+(bb_1)) + \\ & SRS(\mathrm{EETP}^+(bb_3)) + SRS(\mathrm{EETP}^+(bb_5)) + \\ & SRS(\mathrm{EETP}^+(bb_6)) \\ = {} & 20 + 1 + 10 + 20 + 10 = 61\end{aligned}$$

Several synthetic observations over the same path contribute to a synthetic execution time profiles for the unobserved paths $\widehat{\mathrm{EETP}}(\phi_1)$ and $\widehat{\mathrm{EETP}}(\phi_3)$ (dashed paths in Figure 5). Synthetic observations are accumulated on $\widehat{\mathrm{EETP}}(\phi_1)$ and $\widehat{\mathrm{EETP}}(\phi_3)$ until the convergence criterion of the underlying MBPTA process is met, as discussed in Sections II and V.