EPC Enacted: Integration in an Industrial Toolbox and Use Against a Railway Application

Enrico Mezzetti[1], Mikel Fernandez[1], Alen Bardizbanyan[2]
Irune Agirre[3], Jaume Abella[1], Tullio Vardanega[4], Francisco Cazorla[5,6]
[1]Barcelona Supercomputing Center, Spain [2]Cobham Gaisler, Sweden [3]IK4-IKERLAN, Spain
[4]University of Padova, Italy [5]Universitat Politècnica de Catalunya, Spain
[6]IIIA-CSIC, Spain

***Abstract*—Measurement-based timing analysis approaches are increasingly making their way into several industrial domains on account of their good cost-benefit ratio. The trustworthiness of those methods, however, suffers from the limitation that their results are only valid for the particular paths and execution conditions that the user is able to explore with the available input vectors. It is generally not possible to guarantee that the collected measurements are fully representative of the worst-case timing behaviour.**

In the context of measurement-based probabilistic timing analysis, the Extended Path Coverage (EPC) approach has been recently proposed as a means to extend the representativeness of measurement observations, to obtain the same effect of full path coverage. At the time of its first publication, EPC had not reached an implementation maturity that could be trialled industrially. In this work we analyze the practical implications of using EPC with real-world applications, and discuss the challenges in integrating it in an industrial-quality toolchain. We show that we were able to meet EPC requirements and successfully evaluate the technique on a real Railway application, on top of a commercial toolchain and full execution stack.

***Index Terms*—Measurement-based probabilistic timing analysis; Path coverage; Evaluation**

I. Introduction

Critical real-time embedded systems increasingly support complex functionalities realized in software. Some of the corresponding software functions need to pass extensive conformity-assessment procedures, which include verification in the timing domain. Measurement-Based Timing Analysis (MBTA) approaches are very often used in that context in a number of application domains, including automotive, avionics and aerospace [1]–[3], because of their cost-effectiveness and low intrusiveness in the development practice. Reportedly, MBTA is used even for the highest-criticality applications such as DAL-A in avionics [2].

MBTA aims at computing reliable worst-case execution time (WCET) bounds on the timing behaviour of the target program under all possible execution conditions. In practice, however, the validity of MBTA results is limited to the set of execution conditions that were actually triggered and observed at analysis time. In this respect, the degree of *path coverage* guaranteed during the analysis campaign critically affects the soundness of the results. WCET estimates are in fact only valid for the paths actually observed, and cannot be directly generalized to the entire program [3], [4]. It is therefore left to the user to guarantee that the input vectors used for analysis do exercise the program paths that are relevant to the WCET computation (and not necessarily to functional verification).

Unfortunately, however, reasoning on path coverage (or *path-representativeness*) in timing analysis is very challenging. As achieving exhaustive coverage by test is not practically and economically feasible in the general case, measurement-based methods typically assume that the input vectors needed to exercise the relevant program paths are somehow made available by the user. In practice, however, this assumption turns out to be quite unrealistic: the very concept of relevant path in WCET computation is unclear and there is no available metric (alike to MC/DC in functional verification) for it or heuristic supported by scientific arguments. That is, in some sense, as fuzzy as the well-known industrial practice of inflating the highest value observed with "safety margins for the unknown".

Measurement-Based Probabilistic Timing Analysis (MBPTA) [5] is a probabilistic variant of MBTA that allows attaching quantitative confidence to the analysis results, provided that certain conditions are met. MBPTA computes probabilistic WCET (pWCET), representing bounds to the program's execution time, which can be exceeded only with a given probability. MBPTA builds on a mixture of randomization and upper-bounding (worst-case mode enforced at analysis time) to ensure that most hardware-level execution time variability is transparently captured in the analysis observations. As observed in [6]–[8], however, path representativeness is strictly related to the input vectors used for the analysis, which remain under direct user control. The coverage quality of the input vectors needs to be explicitly taken into account to guarantee that the timing behaviour observed at analysis time is representative of the system behaviour during operation (that is, it exhibits all phenomena of interest to capture).

In the context of MBPTA, the Extended Path Coverage (EPC) approach [7] offers an elegant solution to address the path representativeness problem. EPC builds on top of standard MBPTA and improves the representativeness of its results: the concept of probabilistic path-independence is exploited to automatically generate a set of synthetic observations that complement the set of measurements collected. This extended set of measurements, when fed to MBPTA, produces the same effect of *full path coverage* without requiring any additional

978-1-5090-5269-1/17 $31.00 © 2017 IEEE

test effort to the user. MBPTA and EPC addresses the timing behaviour of programs in isolation (i.e., not explicitly factoring in the effects of preemption or multicore contention).

In this work, we focus on the EPC requirements, from the standpoint of the applicability of the technique in realistic use scenarios. In fact, EPC relies on the availability of structural and timing information, whose provision translates into a set of requirements at both hardware and software level. While a prototype implementation and an initial evaluation of EPC had already been presented in [7], they were performed on a hardware simulator using synthetic benchmarks. Hence, EPC requirements still need to be assessed in a real world scenario and against a representative industrial application. In this work we aim at filling this gap by discussing the challenges we addressed in furthering the EPC technique from a tentative prototype to an industrial-quality utility. We report on our experience in providing hardware and software support for EPC on a real hardware platform and an automated infrastructure comprising a commercial timing analysis tool. We show that EPC can effectively be brought far past the prototype stage, and successfully demonstrated it on an industrial application.

The reminder of the paper is organized as follows: Section II introduces the necessary background on MBPTA and EPC; Section III discusses EPC requirements and the challenges we encountered in meeting those requirements in a real setting; Section IV reports on the evaluation of EPC on both a controlled experiment and a critical application from the railway domain. Finally, Section VI draws some conclusions and suggests future work.

II. Background

The MBPTA approach [5] applies the powerful statistical tools of extreme value theory (EVT) [9], historically used for forecasting rare events (e.g., floodings), to software programs. EVT is applied to a set of execution time observations, typically end-to-end measurements, to derive trustworthy upper-bounds to the probability that the execution time of a given program on a given platform may exceed a given (pWCET) threshold. The representativeness of the MBPTA results is warranted by ensuring that the timing behaviour of the hardware components at analysis time exactly matches or upper-bounds [10] their execution time under all possible operation scenarios [11]. Whenever this is not possible, time randomization can be used to probabilistically characterize certain hardware features and probabilistically upperbound [12] their impact on the execution time of a program. Time-randomized caches [13], with random placement and replacement policies, fall in the latter category and have been shown to facilitate the application of MBPTA [14]. Random caches also facilitate incremental software integration [15] as the WCET estimates derived at the unit level account for the impact of any memory layout changes that may happen upon integration. This allows preventing late detection of timing faults and costly corrections of timing budget [15]. An aspect of representativeness that MBPTA cannot guarantee by construction is the one related to the analysis inputs. As any other measurement-based approach, MBPTA can only provide results for the subset of paths triggered by the analysis inputs. Hence, being unable to guarantee that the provided inputs have triggered all relevant paths in the program diminishes the confidence in the resulting pWCET estimates.

The Extended Path Coverage (EPC) technique [7] provides a technical solution to increase the confidence on MBPTA results by guaranteeing the same degree of representativeness as obtainable with full path coverage. In a nutshell, EPC improves the representativeness of MBPTA results by synthetically extending the set of measurements that are fed to EVT and used to compute the pWCET distribution. The addition of synthetic time traces is done in a way to achieve the same effects of full path coverage as it would be guaranteed by exhaustively extending the set of input vectors. Figure 1 outlines the interaction between EPC and MBPTA and the way EPC operates to increase the representativeness of MPBTA results.

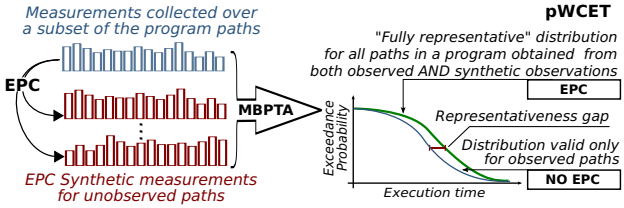

Fig. 1. EPC in relation to standard MBPTA process and effects on its results.

The main requirement of EPC consists in the availability of a set of measurements for each basic block (hence requiring basic block coverage[1]), irrespective of the path leading to it. In fact, to derive synthetic measurements, EPC focuses at the level of basic blocks and exploits the probabilistic nature of time-randomized hardware [13], [17] to derive *probabilistic path-independent* execution times for each basic blocks. Each basic block is likely to exhibit different timing behaviour depending on the execution history, that is the specific path leading to it. In this work we focus on time-randomized hardware architectures [13], where time-randomized caches are the main sources of (intra-core) variability on the execution time of a basic block. Consequently, EPC makes the execution times of basic blocks path-independent simply by probabilistically summing up a penalty or padding each observed execution time, to compensate for any positive cache effect arising from a specific path provenance. To this end, EPC considers the contribution of each memory access to the execution time of a basic block and formalizes the notion of Access-Time Profile (ATP) for each memory access as follows:

$$ATP(@_A, \phi) = \left\langle \begin{array}{cc} L_{hit} & L_{miss} \\ P_{hit}(@_A, \phi) & P_{miss}(@_A, \phi) \end{array} \right\rangle \quad (1)$$

Equation 1 simply models the fact that with time-randomized caches the latency (either L_{hit} or L_{miss}) incurred by an access to a memory location $@_A$ along a path ϕ follows the

[1]Basic block coverage represents a common coverage requirement in DO-178C already from DAL C [16].

probability distribution of that access to be a cache hit or a cache miss. 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. Block execution times are made path-independent by adding a penalty ($L_{pad} = L_{miss-hit}$). L_{pad} compensates for potential cache hits that $@_A$ can benefit along just a subset of the paths leading to it. This helps derive a padded $ATP^+(@_A)$ that tightly over-approximates the theoretical worst-case $\overline{ATP}(@_A)$ along any path in the program, as graphically explained in Figure 2.

The $ATP^+(@_A)$ is obtained by adding a (scalar) padding with a given probability to memory accesses in $ATP(@_A, \phi)$ according to the following definition.

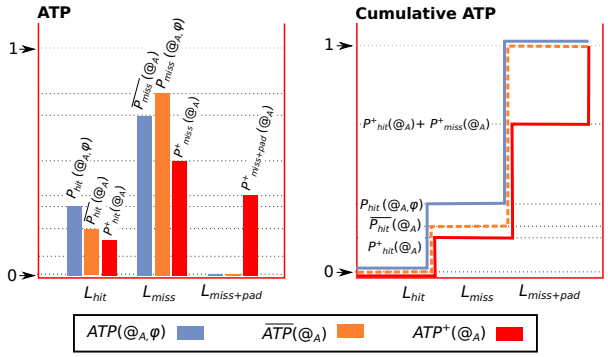

Fig. 2. Relation among $ATP(@_A, \phi)$, $ATP^+(@_A)$ and $\overline{ATP}(@_A)$.

$$
\begin{aligned}
ATP^+(@_A) &= ATP(@_A,\phi) \otimes \left\langle \begin{array}{cc} 0 & L_{pad} \\ 1-P_{pad}(@_A,\phi) & P_{pad}(@_A,\phi) \end{array} \right\rangle \\
&= \left\langle \begin{array}{cc} L_{hit} & L_{miss} \\ P_{hit}(@_A,\phi) & P_{miss}(@_A,\phi) \end{array} \right\rangle \otimes \\
&\qquad \left\langle \begin{array}{cc} 0 & L_{pad} \\ 1-P_{pad}(@_A,\phi) & P_{pad}(@_A,\phi) \end{array} \right\rangle \\
&= \left\langle \begin{array}{ccc} L_{hit} & L_{miss} & L_{miss}+L_{pad} \\ P^+_{hit}(@_A) & P^+_{miss}(@_A) & P^+_{miss+pad}(@_A) \end{array} \right\rangle
\end{aligned} \tag{2}
$$

To enforce this modified distribution, a padding L_{pad} is added to each memory access $@_A$ along a path ϕ, according to the probability P_{pad}, as shown in Equation 2. As formally described in [7], the only constraint is that:

$$
P_{pad}(@_A,\phi) \geq 1 - \frac{\overline{P_{hit}}(@_A)}{P_{hit}(@_A,\phi)} \tag{3}
$$

where $\overline{P_{hit}}(@_A)$ is a lower bound to the hit probability of $@_A$ along any possible path and $P_{hit}(@_A, \phi)$ is the exact probability of $@_A$ to be a hit along path ϕ. To compute $\overline{P_{hit}}(@_A)$ and $P_{hit}(@_A, \phi)$ we exploit the concepts of *reuse distance* and *unique accesses*.

Definition (Reuse distance - *rd*)**.** The *rd* 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 $@_A$. The worst-case reuse distance (along any path) for access $@_A$ is denoted by $\overline{rd}(@_A)$.

Definition (Unique accesses - *un*)**.** With *unique accesses* 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 of $@_A$ on a path ϕ.

Using reuse distance and unique accesses we compute $\overline{P_{hit}}(@_A)$ and $P_{hit}(@_A, \phi)$:

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

where

$$
\overline{P_{miss}}(@_A) = \begin{cases} 1 - \left(\frac{w-1}{w}\right)^{\overline{rd}(@_A)} & \text{if } \overline{rd}(@_A) < w \\ 1 & \text{otherwise} \end{cases} \tag{5}
$$

and

$$
P_{hit}(@_A,\phi) \leq uP_{hit}(@_A,\phi) \tag{6}
$$

$$
uP_{hit}(@_A,\phi) = \begin{cases} 1 & \text{if } un(@_A,\phi) < w \\ \left(\frac{w-1}{w}\right)^{un(@_A,\phi)-w+1} & \text{otherwise} \end{cases} \tag{7}
$$

where w is the number of ways in a set-associative cache.

In our implementation, we over-approximate both $\overline{P_{hit}}(@_A)$ and $P_{hit}(@_A, \phi)$ by computing $\overline{rd}$ and un on a restricted scope (i.e., not considering all execution history backwards). This has been empirically proved to guarantee sufficiently precise results in most cases, at low computational costs.

The values computed for $\overline{P_{hit}}(@_A)$ and $uP_{hit}(@_A, \phi)$ can be used in Equation 3 to compute a sound value for $P_{pad}(@_A, \phi)$, as shown in Equation 8.

$$
P_{pad}(@_A,\phi) = \begin{cases} 0 & \text{if } uP_{hit}(@_A,\phi) = 0 \\ 1 - \frac{\overline{P_{hit}}(@_A)}{uP_{hit}(@_A,\phi)} & \text{otherwise} \end{cases} \tag{8}
$$

To obtain path-independent observations for a basic block *bb* we therefore augment each collected measurement *Obs* by adding a padding for each memory access and path, according to the computed $P_{pad}(@_A, \phi)$. Therefore for all accesses $@_A$ along path ϕ in *bb*:

$$
Obs^+(bb) \mathrel{+}= \begin{cases} Obs(bb,\phi) + L_{pad} & \text{if } rand() \leq P_{pad}(@_A,\phi) \\ Obs(bb,\phi) & \text{otherwise} \end{cases}
$$

These path-independent figures are then combined together to form synthetic end-to-end time traces for each non-observed path $\overline{\phi}$ in the program. The use of path-independent basic block execution times ensures that the constructed collection of values for $\overline{\phi}$ is a valid upper-bound of any collection of real measurements. It is worth recalling that (probabilistic) path-independence is a *necessary condition* as we cannot simply sum the execution time observations over basic blocks (even maxima values) to obtain the execution time of unobserved paths. The main reason is that observations are only relative to a particular path traversal and its particular cache-level and core-level timing dependencies[2].

From a procedural standpoint, EPC operates on MBPTA inputs and therefore it does not require any modification to the way MBPTA uses EVT in the analysis process. However, in order to generate an extended set of measurements, EPC relies on the availability of basic-block level timing information and

[2]Timing anomalies, which we do not consider here as they do not appear in our reference platform, would make this naive approach even more unsound.

basic knowledge on the structure of the program, which are not part of the standard MBPTA process. In fact, MBPTA operates as a black-box process where all representativeness issues are left to the user. The collection of the structural and timing information is necessary for EPC to: (i) understand basic block boundaries; (ii) reconstruct memory accesses to compute the reuse distance and unique accesses; and, (iii) find unobserved (but still feasible) paths. Whereas the required information may be simple to collect in a synthetic and controlled environment [7], it poses some requirements when moving to realistic setting and applications. In the following section, we discuss the challenges we had to face when trying to meet EPC requirements in an industrial setting, and analyze a representative industrial application.

III. Improving representativeness

EPC strengthens the quality of MBPTA results by extending the analysis inputs to achieve the same degree of representativeness of full path coverage. Whereas MBPTA inputs typically consist in coarse-grained timing traces, EPC relies on additional information as well as adequate mechanisms for this information to be efficiently collected. The entailed complexity became evident when we considered to move EPC from an early-stage prototype running on a simulator, to an industrial-quality toolchain executing on top of a real platform. In the following we report on our experience in implementing EPC as a plug-in of the RVS[3] analysis suite and targeting a time-randomized FPGA design [13], based on the LEON3 family of processors[4]. To this extent we first single out the peculiarities and requirements of EPC process when compared to standard MBPTA and then provide a technical discussion on how those requirements were effectively met.

A. EPC process requirements

When considering the overall timing analysis process, MBPTA can be unfolded into four main phases: (i) *preparation* of the target program (ii) *collection* of traces; (iii) *processing* of traces; and (iv) final *computation*. Table I itemizes the EPC requirements and classifies them according to the part of the MBPTA process that they pertain to, as well as to the type of hardware (HW) or software (SW) support that they call for. In the remainder of this section, we first discuss each of these requirements and then illustrate how we met them in our implementation.

The preliminary ***preparation*** phase is responsible for setting up instrumentation for trace collection and building the executable. Instrumentation can be performed with hardware or software means. The former is in general transparent so that it causes no overhead on program execution time, but requires specialized hardware. The latter, which we use in our framework, represents the most generic and portable solution. As a known drawback, software instrumentation affects timing (i.e., *probe effect*). Deploying the instrumented code could penalize performance, while removing it may be difficult to justify against stringent industrial qualification and certification standards. Recent work [18] shows how to exploit *nop* operations to substitute instrumentation instructions in a way that simplifies qualification/certification and at the same time reduces the impact of instrumentation. To support EPC, instrumentation is necessarily applied at basic block level: automatic support to instrument and build an instrumented version of the target executable is required (**R1**).

TABLE I
EPC requirements broken down into hardware and software

Id	Phase	Type	Requirement
R1	*Preparation*	SW	Instrumentation at basic block level
R2	*Collection*	HW	Tracing throughput adequate to fine-grained software instrumentation
R3	*Collection*	HW	Collection of memory accesses information
R4	*Collection*	HW	Collection of information on random seeds for memory placement
R5	*Processing*	SW	Augmented empirical execution time profiles
R6	*Processing*	SW	Generation of synthetic observations

The ***collection*** phase includes gathering timing traces for each input vector (and thus for each path). The set of input vectors should provide full basic block coverage. Note that basic block coverage is less demanding than the most common coverage metrics widely adopted in functional testing. Basic block level instrumentation, however, requires largest throughput to be supported by the trace infrastructure (**R2**). Moreover, the collection phase is where all the data required by the analysis is gathered. With EPC the set of data is not limited to timing traces, but also includes structural information on the set of data and instruction addresses within each basic block as it is required to properly compute EPC probabilistic padding. In fact both reuse distance (*rd*) and number of unique accesses (*un*), as defined in Section II, are determined by combining information on memory accesses and how they map to the cache, which in turns depends on cache placement and replacement functions. Although this kind of information could be statically computed from, for example, information derived from the compiler, we decided to derive it from a dynamic analysis step. It consists in the collection of address and instruction traces with an additional run for each observed path, solely seeking structural information, without considering timing. In contrast with timing traces, where several runs of each path are required to characterize their timing behaviour, *one single run* is sufficient to collect an *address trace* with all the required structural information. As a result we need to derive and store information on both cache mapping and memory accesses. In time-randomized cache, cache mapping depends on the current seed used to determine random placement and replacement functions: while the effect of random replacement is transparently captured by the timing traces, the seed used to determine the random placement function is crucial in the computation of *rd* and *un*. We therefore need efficient means to collect memory accesses for data and instruction (**R3**) and information on the random placement function (**R4**).

In the ***processing*** phase, the collected information is un-

[3]Rapita System Ltd., https://www.rapitasystems.com.

[4]Cobham Gaisler, LEON3 IP Core, http://www.gaisler.com/index.php/products/processors/leon3.

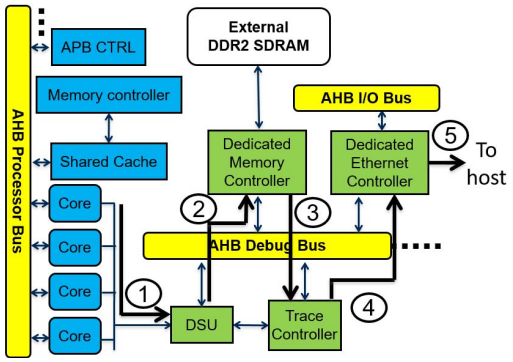

Fig. 3. Schematic of the Ethernet Tracing feature and steps to output traces.

derstood and organized in a way that allows later use in the analysis, which amounts to building an augmented control flow graph (CFG) representation of the target program (**R5**). As specific steps in EPC, the collected structural and timing information is exploited to compute the probabilistic padding as well as to find the set of unobserved paths. Automated and computationally efficient tool support is needed to generate the set of complementary synthetic measurements (**R6**).

EPC does not pose any requirement on the final ***computation*** phase. In it, we proceed exactly as in standard MBPTA, with the only difference that, owing to the extended set of inputs that we provide, the results are representative of a larger (ideally exhaustive) set of paths. Real and synthetic traces are fed to the EVT tool box and a worst-case distribution is computed and used to derive pWCET figures for the thresholds of interest.

B. Hardware-level requirements

The hardware platform to which EPC can be applied must provide a number of features. Some of those features are transitively derived from MBPTA prerequisites while other are specific to EPC. In particular, EPC needs the processor to be made MBPTA-compliant using random modulo placement and random replacement for cache memories [13]. Such hardware features, along with other modifications to attain MBPTA compliance, have been implemented in an FPGA prototype [19] combining components from LEON3 and LEON4 architectures developed by Cobham Gaisler, and extended to match the timing properties required by MBPTA and inherited by EPC. To further support the additional requirements set by EPC, the hardware platform has been augmented with a number of tracing and interfacing capabilities.

R2 *- Tracing throughput adequate to fine-grained software instrumentation:* The tracing support originally available on the board consisted on the capability of outputting pairs identifying instrumentation points and timestamps through the GPIO pins. However, this showed severe drawbacks: it only allowed reading traces at a very low speed; limited the instrumentation points to 256 (only 8 bits available per core); and caused stalls in the execution when instrumentation instructions were over-saturating the output.

To overcome these limitations, a new tracing solution has been implemented on top of a decoupled Ethernet tracing scheme, as shown in Figure 3. First, instruction and data addresses (if any), opcode and timestamp of all instructions are output through the debug interface (DSU) not to interfere with program execution. A dedicated memory controller stores only the instruction address and timestamp of timing instrumentation instructions in a private external DDR2 SDRAM region, thus decoupling trace collection from trace outputting. Then, a dedicated trace controller reads asynchronously those traces and outputs them through a dedicated high-speed Ethernet port. Since the Ethernet port works at high speed, its pins are not devoted to specific cores and thus, an arbitrary number of bits can be used to encode instrumentation points. This tracing mechanism allowed producing all timing traces without interfering with the execution of the application.

R3 *- Collection of information on memory accesses:* The very same hardware feature for timing trace collection is used for address trace collection. The mechanism is analogous but, instead of storing in SDRAM information for instrumentation instructions, it is configured to record all data sent through the DSU for all instructions. Since the amount of information to be collected is much higher than in the case of timing traces, trace dumping to the host may not keep pace with the execution of the program being traced. However, since address traces are collected independently from the timing information, whenever the tracing feature cannot keep pace with the execution of the program, execution is stalled to not lose any information.

R4 *- Collection of information on random seeds for memory placement:* The particular seed used for random modulo placement in caches can be intercepted by software means. We identified two ways to collect those seeds: either recording them whenever they are set or obtaining them from hardware. The implied software process is only slightly different. Since software layers may not allow connecting the module setting the seed with the tracing features in place, we collect the seeds by reading the specific register where they are mapped. We use the GRMON monitor to access the specific memory location where the seed is stored right before the execution of the program and dump it into a file. While this interface is slow, it only needs to be used once per run and right before the execution of the program, thus not affecting timing.

Applicability to other platforms: The hardware framework and solutions implemented and evaluated in this work do not necessarily represent the only possible way to enable the industrial use of EPC.

For (time) tracing, EPC does indeed require non-negligible throughput capabilities. Basic block level instrumentation produces a larger amount of timing information than coarser-grained hybrid instrumentation approaches. The solution we adopted, while specific to the processor we used, does not rely on ad-hoc HW features, but simply makes smart use of standard tracing facilities (DSU, Memory and Ethernet controllers) already present in the LEON processor family. Other processor architectures either already provide or are introducing similar features, in response to stringent market requests. Advanced tracing functionalities, such as the Nexus Interface [20] for

NXP (formerly Freescale) or the Coresight [21] for ARM, are increasingly being considered for measurement-based and static timing analysis solutions [22], [23].

Besides timing information, EPC also relies on the availability of a detailed characterization of instructions and data accesses within each basic block, as memory accesses are required in the computation of the probabilistic padding. Both static and dynamic methods can be adopted to derive this kind of information. In our implementation we opted for a dynamic approach and took advantage of the native support provided by the LEON AHB bus interface that allows full access to the bus traffic. As already discussed, the resulting timing interference is of no consequence to the analysis, as we are not considering the timing aspect in this step. The fact that a given architecture does provide support for finer-grained snooping is not preventing us from collecting the same information by exploiting standard compiler information and data-flow analysis. The complexity of statically deriving information on memory accesses is generally acceptable (considerably less than end-to-end static timing analysis) and is supported by standard timing analysis tools.

Finally, even the adoption of time randomized caches in hardware is not really mandatory. Indeed, time-randomized cache behaviour is a key assumption to EPC and characterizes the way padding is computed and applied to the timing measurements at basic block level. However, analogous randomization effects can be obtained by means of specific software libraries, transparent to the application, as shown in [17]. Applying EPC to a software-randomized setting should only require minor modifications to the padding computation libraries. This is part of our future work.

C. Software-level requirements

To be usable industrially, EPC must be accompanied with adequate software infrastructure support for all the tasks entailed in the analysis. We built the EPC process on top of the RVS analysis suite and its suite of tools and utilities, on account of it being a DO-178C-certified [16] tool set for timing verification of embedded systems. The EPC implementation thus relies on standard RVS toolchain elements and custom tools that use the RVS APIs. The resulting toolchain provides support for the whole EPC process and meets the technique requirements, as they were highlighted in Section III-A.

R1 *- Instrumentation at basic block level:* EPC works on timing information at the granularity level of single basic blocks. Ideally, the analysis should be performed on object-level basic blocks, which may differ from the source-level ones due to completely transparent compiler transformation and optimizations. The RVS suite already provides fully automated support for pre-processing and instrumenting the source code that can be easily integrated into any industrial software development environment.

Source-level basic blocks are the most fine-grained level of abstraction supported by RVS instrumentation and timing data manipulation libraries. To preserve the applicability of EPC, we forced the application to be compiled with no aggressive compiler optimizations so that source- and object level basic blocks coincide. There is no theoretical impediment to facing compiler optimizations as long as the toolchain is able to reconstruct the program structure from the binary. Static analysis industrial-quality tools typically offer this functionality.

R5 *- Augmented empirical execution time profiles:* Timing data need to be stored along with information on context of execution and on the random placement seed in use when the measurement was collected. Both aspects are in fact crucial in the computation of the probabilistic padding in that they determine both the set of basic block predecessors and the tightness of random cache metrics. RVS toolchain already organizes timing data in a data structure shaped on the program CFG. RVS operates on empirical execution time profiles, where basic blocks execution times are associated with their respective frequency of observation.

This organization is particularity convenient for EPC. We therefore exploited the RVS framework and extended its core data structure to hold all the extra information specific to EPC. Currently, the contextual information is limited to the set of immediate predecessors in the CFG and improved with limited virtual loop unrolling. Based on preliminary experiments, this has been considered to provide enough precise information for a first implementation. Extending the framework to hold more information on execution context is part of our future work.

In order to compute and apply EPC probabilistic padding to execution time profiles we implemented an external library based on the RVS API that could be called at the appropriate time by standard tools in the RVS toolchain.

R6 *- Generation of synthetic observations:* In order to produce the complementary set of synthetic observations, EPC relies on knowledge of observed and unobserved paths[5]. The complexity of providing automated support to this step of EPC was not made evident in [7], owing to the prototype nature of the tool and its ad-hoc evaluation. Discovering all paths in a program may easily hit the complexity wall. The number of paths grows exponentially with the number of conditional branches and loops, which makes the path enumeration problem poorly scalable and already intractable with relatively simple programs. However, it is also true that not all *structurally feasible* paths are always feasible in practice, due to the combination of execution conditions. The set of *semantically feasible* path is typically much more manageable. Based on this observation, we developed a tool that support semantic information and relies on an efficient program model to explore and generate a large number of paths, without compromising efficiency in time and space.

This internal model is built through a set of model-to-model transformation from a tree-based representation of the program CFG that is already present in RVS. The model is augmented with information on the maximum observed loop bounds (used to further reduce the search space) and with support for simple control-flow constraints to refine the

[5]Collecting synthetic measurements for already observed paths would introduce unnecessary pessimism.

semantics of the program. The use of semantic information (*flow facts*) to narrow the number of paths to be considered in the analysis is a well-known approach, for example, in static timing analysis techniques [24]. The basic support to flow facts currently implemented allows to defining maximum iterations of loops and correlations between conditional constructs.

Our tool only intercepts unobserved paths as the set of observed paths is automatically derived from the timing information in the RVS core data structure. The optimized program model allows the tool to perform also full path enumeration in just a few seconds, even for real-scale industrial programs. Synthetic observations for unobserved paths are generated by sampling timing data from the RVS data structure [7].

IV. Evaluation

In general, providing a qualitative and quantitative evaluation of the precision of measurement-based timing analysis approaches is a complex endeavour. In realistic scenarios, the complexity of the analyzed application does not typically offer a-priori knowledge of its worst-case timing behaviour (and the execution path leading to it). The focus of our evaluation is on comparing the EPC results with maximum observed execution times and plain MBPTA results. Assessing MBPTA (and EPC) against static deterministic approaches falls outside the scope of this work. Interested readers can refer to [25] for a preliminary comparison and a discussion on the complexity of making a fair comparison.

The results obtained with EPC necessarily dominate those obtained with standard MBPTA. As EPC augments the MBPTA input space, it is always guaranteed to produce pWCET distributions that are at least equal to those obtained from real measurements. Results become representative of all possible paths in the program, even of those that have not been traversed during measurements and whose timing behaviour is unknown. The results of EPC, however, also include a certain degree of over-approximation as a consequence of the conservative application of probabilistic padding. Hence, when looking at the pWCET profiles obtained with EPC against those produced by standard MBPTA, it is difficult to tell apart the contribution of the extended path coverage from the effect of overestimation.

A numerical evaluation of EPC on an heterogeneous set of synthetic benchmarks has been already presented in [7]. That evaluation, however, was performed on a prototype implementation running on a hardware simulator, as part of a research toolchain. In this work instead we present an evaluation of a solid implementation of EPC in an industrial-quality setting against a real-world application. As the original evaluation in [7] did not look into what factors contributed to the EPC results, in this work, we also study: (i) the extent to which the EPC results are determined by better representativeness as opposed to overestimation, and (ii) to what extent the incurred pessimism relates to path coverage (i.e., how many synthetic paths are considered in the computed worst case). To assess the latter, however, we need to have full control over the coverage of the program paths, which is generally too complex in real-world applications. For this reason, before moving to the evaluation of our technique on an industrial case study, we present an evaluation on a synthetic controlled experiment.

A. Controlled experiment

The first objective in our evaluation was to gain better insight on the precision of the EPC results (i.e., tightness with respect to the actual worst-case path), especially considering whether and to what extent path coverage affects the final EPC results. Reasoning on precision, however, is impaired by the fact that, for a given target program, we typically do not know the actual WCET (and path leading to it). We equally cannot try to derive it as an afterthought by forcing the execution of the worst-case path identified by EPC since controlling all aspects of execution through input parameters may be exceedingly hard, owing to the complex interaction of input vectors with internal software/hardware states.

We therefore put together a synthetic demonstrative example, small enough to be easily managed but at the same time not trivial, to experiment the technique. This example uniquely aims to let the reader have a feel of EPC's precision (i.e., improved representativeness against overestimation) in relation to the initial set of observations. This discussion therefore should not be intended as an exhaustive quantitative evaluation of the EPC results, for which the results reported in [7] should suffice. For these reasons, we kept the illustrative example as simple as possible to allow for a clear mapping between input vectors and execution paths.

The example, which we call `MultiBranch`, consists in a single procedure exhibiting four `if-then-else` constructs in a sequence (thus yielding a total of 16 paths). Each branch performs some computations and executes some loops with a different number of iterations so that each path is expected to exhibit a different timing behaviour. An overview of the structure of `MultiBranch` is provided in Figure 4. The program structure is represented by a simplified control flow graph, where basic blocks are conveniently grouped to highlight conditional branches. A representation of the possible paths is also reported (smaller diagrams on the right), where darker nodes represent program parts that belong to that specific path.

The outcome of each branch decision is explicitly controlled by a value in the input vector of `MultiBranch` so that we can generate 16 input vectors and trigger 16 different paths, by simply operating on 4 input parameters. In this way we are able to exercise full control on the program's behaviour and identifiy the minimum number of paths – that is just the two paths framed in red in Figure 4 – required to fulfil EPC coverage requirement. Knowledge on the program semantics also allowed us to identify a priori the worst-case path (i.e., leading to the worst-case behaviour), framed in black in Figure 4. Full control on branch decisions, in conjunction with the limited number of feasible paths, allowed us to collect exhaustive measurements for all paths in `MultiBranch`. Note that the sets of structurally and semantically feasible paths coincide.

We focused on five different scenarios of application of EPC with varying number of observed paths:

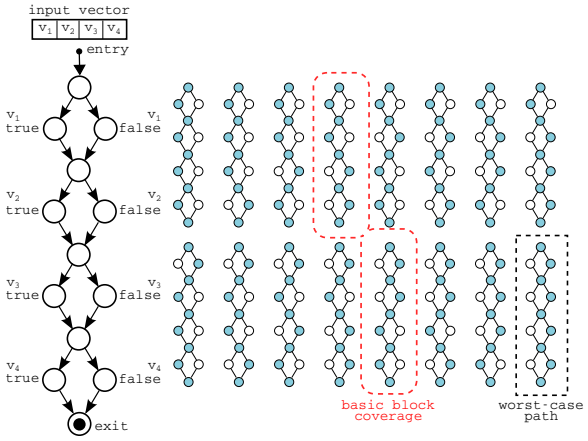

Fig. 4. Overview of `MultiBranch` structure.

- `16paths` All paths triggered at analysis time so that measurements are already fully path-representative;
- `8paths` Half of the paths actually measured, excluding the worst-case path.
- `4paths` Four paths actually measured, excluding the worst-case path.
- `2paths` Only two paths actually measured (providing basic block coverage), excluding the worst-case path.
- `W-8paths` Half of the paths actually measured, in this case including the worst-case path.

Under each configuration we applied MBPTA on all program paths separately. Among all paths we then selected the maximum pWCET for the probability thresholds we were interested in. For each path we collected (or generated, for synthetic paths) 2000 runs that were sufficient for MBPTA to converge. The so-obtained set of inputs successfully passed statistical independence and identical distribution tests so we could apply EVT.

TABLE II
MBPTA RESULTS FOR THE FULL PATH-COVERAGE SCENARIO (MULTIBRANCH).

Path		MOET	10^{-03}	10^{-06}	10^{-09}	10^{-12}
16paths	P1	5420	5642	5781	5995	6210
	P2	5777	6011	6165	6399	6633
	P3	5649	5887	6045	6275	6512
	P4	5727	5952	6105	6322	6547
	P5	5621	5694	5774	5897	6020
	P6	5543	5765	5922	6150	6385
	P7	5424	5670	5823	6067	6304
	P8	5383	5427	5490	5580	5676
	P9	5528	5730	5856	6052	6241
	P10	5235	5467	5628	5862	6097
	P11	5844	6096	6259	6498	6736
	P12	5972	6164	6308	6512	6723
	P13	5714	5758	5823	5918	6012
	P14	5959	6229	6404	6655	6913
	P15	5193	5407	5554	5767	5988
	P16	6152	6395	6551	6787	7022

Focusing on the `16paths` scenario, Table II reports the Maximum Observed Execution Time (MOET) and pWCET estimate for each individual path. The pWCET estimate, which is shown at relevant thresholds, is computed by MBPTA over the set of measurements obtained by exhaustively observing all paths in `MultiBranch`. As expected path number 16 (P16) corresponds to the worst-case path. We used MOET and pWCET values obtained in the `16paths` scenario as the baseline for comparison of all the other scenarios. By having the MBPTA results on all observed paths, we can reason on both the un-representativeness (and un-safeness) of partial coverage results and the overestimation incurred by EPC.

TABLE III
MBPTA RESULTS FOR PARTIAL COVERAGE PLUS EPC SCENARIOS (MULTIBRANCH).

Test	MOET	%	10^{-3}	%	10^{-6}	%	10^{-9}	%	10^{-12}	%
2p	5777	-6.1	6018	-5.9	6179	-5.7	6398	-5.7	6654	-5.2
EPC	6917	+12.4	7254	+13.4	7496	+14.4%	7855	+15.7	8214	+17.0
4p	5727	-6.9	5945	-7.0	6083	-7.1	6300	-7.2	6511	-7.3
EPC	7021	+14.1	7337	+14.7	7604	+16.1	8009	+18.0	8384	+19.4
8p	5844	-5.0	6011	-6.0	6165	-5.9	6399	-5.7	6633	-5.5
EPC	6878	+11.8	7180	+12.3	7445	+13.6	7837	+15.5	8238	+17.3
W-8p	6152	+0.0	6395	+0.0	6551	+0.0	6787	+0.0	7022	+0.0
EPC	6987	+13.6	7369	+15.2	7644	+16.7	8056	+18.7	8369	+19.2

Table III shows the results observed in the remaining four experimental scenarios. The largest MOET and pWCET are reported for each threshold and compared against the respective values in the `16paths` scenario, by reporting the difference in percentage. Several interesting observations can be drawn from the experiments. First, results confirms that EPC delivers fully representative pWCET figures that over-approximate the pWCET obtained by actually observing all paths in a program. Conversely, MBPTA results obtained with partial coverage are necessarily flawed when the worst-case scenarios are not captured at analysis time. The degree of overestimation incurred by the current implementation of EPC is approximately 16% in the average, with minimum and maximum at 10% and 19% respectively, when compared to the worst case scenario we were able to detect in the full-coverage controlled experiment. In almost half of the scenarios, applying the 20% margin widely adopted in industry produces more pessimistic results than EPC (although in this particular case it is shown to be effective in over-approximating the worst case). In particular, the pessimism introduced by path-independence and synthetic path generation, as observed in the increase on the MOET, is very limited (+14% in the worst-case). As shown in Figure 5, the residual increase in the pWCET is a direct effect of "moving" the pWCET curve to the right.

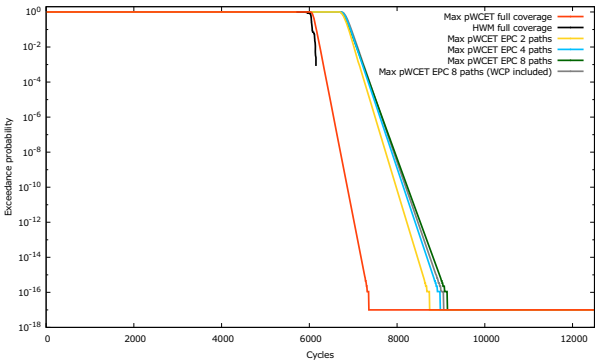

Fig. 5. EVT projections: full-path vs EPC over-approximations.

Notably, the extent of the resulting overestimation does not seem to strongly depend on the degree of coverage before applying EPC. Although it is not appropriate to draw general conclusions from a single small example, we deem this experiment to provide solid evidence that synthetic measurements smooth over the possible execution-time differences among program paths. This is in part explained by the fact that synthetic measurements are not always identical as they sum up random effects from basic block execution times and therefore their contribution to the pWCET is neither constant nor easily predictable. Interestingly, this means that basic block coverage is really the *only* test requirement stemming from EPC. The results from the `W-8paths` scenario show that the worst-case values after applying EPC are greater than the real (measured) ones. This indeed may happen when the difference between the worst-case and other values is not so large that it can be flattened by EPC (in fact it lays beyond the 14% average increase on MOET).

B. Industrial case study

Subsequently, we evaluated our EPC implementation on a real application from the railway domain. The trial application is a simplified European Train Control System (sETCS) railway signaling subsystem. The "simpified" tag should not mislead the reader: the sETCS is a ***faithful*** reimplementation of a real train control application, developed by the same senior engineers that implemented the original system, which is currently in use. Hence, while the application we experimented with is not the original one, it is fully equivalent to it for execution semantics, timing behaviour, and structural complexity.

The sETCS is a safety-critical application (SIL 4) that keeps train motion under control. Essentially it supervises the traveled distance and speed, and activates the brakes if authorized values are exceeded. The central safety processing unit of the ETCS system is called European Vital Computer (EVC) and, for safety purposes, it is implemented by three parallel implementations of the same logic (i.e., Triple Module Redundant - TMR computing nodes). At a lower level, each computing node supports an instance of the safety software application, which comprises three main tasks that are sequentially executed: (a) Odometry module (OMS): is the responsible of estimating a set of parameters based on the information received from the train environment (e.g., estimated train position); (b) Emergency module (ES): controls the Emergency braking system; and (c) Service module (SS): controls the Service braking system.

In our evaluation we focused on the ES module as the core of the safety-critical elements of sETCS. The ES logic includes several data-dependent paths. The analyzed module has been automatically generated from a Simulink model and includes more than 50 conditional branches (few of which are switch constructs), 90 procedures (for approx. 330 procedure calls) and nearly 120 basic blocks. Experiments were run at the user's premises and the application was made available along with an extremely limited set of input vectors (just 10) that do not warrant full path coverage. Those input vectors were not identified and constructed for the purpose of our experiments: they were part of the standard functional validation suite of the sETCS and were selected by the industrial user because they warranted the sought basic block coverage.

TABLE IV
BASELINE MBPTA RESULTS (SETCS).

Experiment	MOET	10^{-03}	10^{-06}	10^{-09}	10^{-12}
Test0	107218	107343	107602	107731	107990
Test1	108114	108292	108423	108684	108945
Test2	100595	100937	101301	101665	102029
Test3	100330	100737	101101	101585	102069
Test4	101604	101822	102191	102559	102927
Test5	101089	101313	101557	101922	102166
Test6	108283	108855	109508	110161	110813
Test7	114775	114938	115077	115492	115631
Test8	71955	72319	72928	73450	74058
Test9	68110	68270	68517	68764	69094

The application of MBPTA and EPC has required no modification to the industrial application, except for those needed for software instrumentation. We generated an instrumented version of the program by exploiting the fully automated and transparent framework from the RVS toolchain. The instrumented version is then compiled (with a standard GCC cross-toolchain) into a binary and loaded into the target platform. In order to collect timing measurements (and address traces), the available subset of input vectors, characterizing the train environment, are fed to the hardware platform by an Ethernet communication channel. Our EPC experiment reused the technical assets available at the user premises without asking for any effort to prepare and performs the observation runs. In the experiment setting, each provided input vector did not directly correspond to a given path in the program, but rather contributed to updating the internal software state, which in turn triggers a given path of the ES. Without a profound knowledge of the application – which we did not have and could possibly obtain from the software owner – it was beyond our reach to trigger different paths by solely operating on the provided input vectors.

To establish a reference comparison, we first analyzed the ES module according to the base MBPTA method. Results are obtained by performing the EVT calculation on the set of collected measurements, sorted per path. In this way, we obtain per-path pWCET distributions. We select the maximum pWCET among all observed paths for a given threshold. Since the provided input vectors do not exercise all paths, these figures are only valid for the set of observed paths and no general conclusion on the pWCET of the ES can be drawn.

As shown in Table IV, Test7 is the path that exhibits the worst-case observed end-to-end timing and it is also associated to the highest pWCET value for all exceedance thresholds. Notably, this is not a given, as pWCET values at the given threshold are largely determined by the slope of the pWCET envelope rather than a single (maximum) observed value.

We applied EPC to extend the representativeness of the collected measurements by generating synthetic observations for unobserved paths. As discussed above, the application only came with a very small number of input vectors, drawn

from the consolidated set of input vectors used for functional validation. The input vectors were, however, enough to trigger the 10 different paths needed to meet the basic block coverage requirement of EPC. In order to extend path coverage, we first used our custom path-generator tool to identify the set of structurally feasible paths: a total of 12996 paths were detected by the tool. The user was then asked to provide additional control flow information, in the form of annotations, to restrict the number of paths to be considered by EPC to the set of semantically feasible paths. The few flow facts provided by the user caused our tool to exclude 12970 paths that were recognized to be either semantically infeasible or not relevant to the analysis (e.g., exception handling). Thanks to this pruning, the number of paths to be considered to guarantee exhaustive path coverage was reduced to just 26. Since 10 of them were already covered in the observation runs, EPC was used to generate synthetic measurements for the remaining 16 additional paths. The extended input provided to MBPTA consisted in 5,000 observations for each observed and unobserved path (a total of 130,000 samples), enough for all of them to meet the MBPTA convergence criteria.

TABLE V
EPC RESULTS (SETCS).

Experiment	MOET	10^{-03}	10^{-06}	10^{-09}	10^{-12}
SynthTest0	182438	190255	204396	217864	230995
SynthTest1	179803	187309	200111	211998	223886
SynthTest2	180440	183855	193071	201595	210120
SynthTest3	182977	189679	202229	213848	225468
SynthTest4	182560	190485	202810	214669	226529
SynthTest5	180703	188167	198545	208460	218376
SynthTest6	181257	186042	194645	203014	211151
SynthTest7	182721	189052	201127	212969	224580
SynthTest8	183779	200116	222674	244058	265676
SynthTest9	179049	187896	200389	212427	224692
SynthTest10	179818	183464	192189	200454	208948
SynthTest11	182609	187470	197928	207921	218147
SynthTest12	184026	188449	200485	212049	223377
SynthTest13	179069	182793	191440	199631	208051
SynthTest14	184003	187541	197652	207528	217403
SynthTest15	181357	182770	188802	194602	200170
Increase wrt Worst-Case MBPTA	+60%	+74%	+94%	+111%	+130%

Table V reports the results of applying MBPTA on the extended set of observations. and compares EPC results against the baseline ones. As expected, the extended set of paths obtained with EPC exhibits new maxima. The new MOET (determined by the synthetic Test12) is in fact 184026 cycles, which corresponds to a 60% increase over the 114775 MOET of the original set of observations. When it comes to pWCET figures, the increase of the fully representative results (determined by synthetic Test8), as compared to the baseline ones, is starting at +74% and is increasing with lower exceedance probabilities, owing to the slope determined by synthetic measurements.

When analysing the obtained pWCET distributions, it is difficult to reason on EPC precision by telling apart the contribution of unobserved paths from the effect of conservative assumptions. In contrast with the controlled experiment, we were not able to directly stimulate the synthetic paths discovered by EPC without resorting to ad-hoc modifications to the program source code (which we did not want), since we could not establish direct correlation between inputs and paths.

On the other hand, however, the additional information available as a by-product of EPC allows drawing some conclusions. In particular, we can derive the enumeration of basic blocks on each observed and synthetic path, as well as the baseline and augmented empirical execution time profiles for each basic block. This information can be used to map back the worst-case path computed by EPC on the program control flow structure and roughly reason on tightness.

We first focused on the characterization of SynthTest12, the worst-case path reported by EPC. By construction, we know that SynthTest12 was not in the original set of observed paths. Looking at its specification in term of basic blocks, we observed a strong similarity with Test7, the worst-case path according to the actual observations. By code inspection we concluded that they only differ in a high-level branch decision in the decision part of the control algorithm, which results in an additional procedure call and few more computations in SynthTest12. This confirms that the worst-case path observed at analysis time was not the actual worst case, which is instead singled out by EPC.

Subsequently, we made an attempt to gauge the precision of EPC in the particular case of the sETCS. To this end, EPC-padded data have been compared with the respective baseline. Figure 6 summarizes the effect of applying the probabilistic padding at basic block level. For each basic block we consider the set of (observed and synthetically extended) execution times and report the maximum value before and after padding, represented by the darker and lighter columns respectively.

Notably, not all basic blocks were augmented by the padding. In our experiments, in fact, probabilistic padding was only applied to 24 basic blocks out of 113 in the target application, a mere 23%. In the average case, probabilistic padding only determines a relatively small increase as compared to the baseline value to which padding is applied. Exceptions occur for a limited number of basic blocks (highlighted by light-red columns) where the padding has the effect of determining a maximum observed execution time that is up to 4x the original one (from 645 to 3,445 cycles in the worst case). The average increase, instead, was around 10.5%. The effect of over-pessimistic application of padding depends on the block execution frequency and, finally, on the relative contribution of the basic block to the end-to-end path.

To further reason on EPC precision, we focused on the MOET associated to observed and synthetic paths and in particular to the worst-case ones (Test7 and SynthTest12 in Table V). Although we could not execute the SynthTest12 path on the hardware, it is still possible to reason on MOET tightness by considering the contribution of the single basic blocks to the path timing.

Table VI compares the MOET (the synthetic one for SynthTest12) of the two paths against the timing obtained by respectively summing up the maximum (Max Cumul) and minimum (Min Cumul) observed values, and the maximum

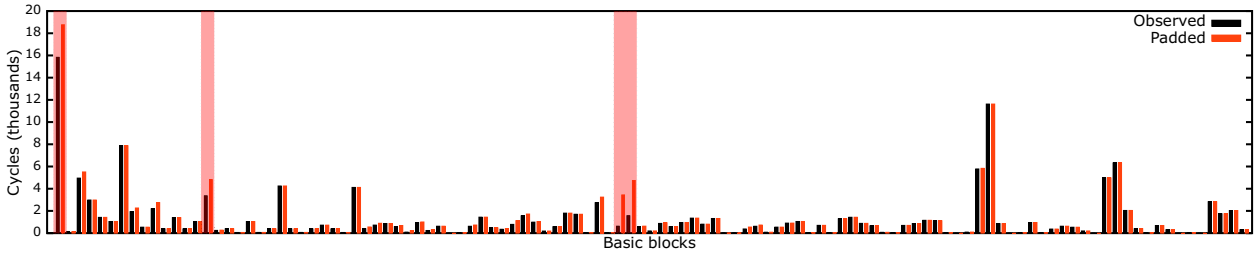

Fig. 6. Original and padded execution times per basic block (sETCS).

TABLE VI
REASONING ON EPC PRECISION.

	MOET	Max Cumul	Min Cumul	Max EPC Inflation
Test7	114775	127282	80105	145282
SynthTest12	184026	163520	92699	191717
Increase	60%	29%	16%	32%

inflated (Max EPC Inflation) execution times for each basic block in the path. The MOET synthetically computed for SynthTest12 is larger that the Maximum cumulative timing and near to the maximum inflation incurred by EPC. This is because the (synthetic) MOET includes potentially unobserved cache effects and is actually computed by summing EPC inflated execution times. For similar reasons, the maximum and minimum cumulative values define a valid range for the MOET only for Test7. In case of SynthTest12, the MOET is necessarily going to be larger than that of Test7 (as we know it is the worst-case path) and smaller than the MAX EPC Inflation. The differences between the two paths seem to suggest that the worst-case path could incur some 30% larger execution time that Test7, which seems to be compatible with what we could determine from code inspection. This means that the synthetic MOET computed by EPC seems to introduce another 30% factor by applying over-pessimistic paddings.

Looking into the source code and binary, we were able to understand that overly-pessimistic padding was generally ascribable to data accesses in case of *dynamic references* and misinterpreted *always-miss accesses*. In the sETCS, dynamic references are used to provide inputs to the core computations: function parameters passed by reference are conservatively considered as dynamic (i.e., unknown) reference although they are following known patterns for each specific execution path. Unfortunately, the current tool infrastructure is not able to either understand or exploit this kind of information. Categorizing them as dynamic accesses dramatically increases the number of applied paddings and negatively affects the probabilities of surrounding accesses. Always-miss memory accesses are those accesses that invariably trigger a cache miss, independently of the specific path. In some cases, the available structural information is not sufficient to intercept those accesses: as a result, we may end up applying a probabilistic padding even when accesses are always-miss, to use a well-known term in static WCET analysis.

In the analyzed application (automatically generated) we found both dynamic references and relatively large sequences of initialization code, whose effects in term of precision were non-negligible and seem to largely affect the overall precision of EPC results.The pessimism incurred in the sETCS case could be cured by extending the computation model to improve the support for contextual information. Although there was no technical impediment to those modifications, it was not possible to incorporate them into the toolchain as they would have required considerable modifications to the consolidated RVS industrial-quality tool. It is certainly part of our future work to further investigate possible extensions to RVS.

V. RELATED WORK

The reliability of measurement-based timing analysis approaches has been repeatedly questioned [3]. The main argument used against these approaches cites the inherent difficulty to provide evidence that the worst-case scenario, in terms of both inputs and hardware state, has been actually observed. MBTA approaches, in fact, only capture what is known as the maximum observed execution time (MOET). The consolidated industrial practice is then to add an engineering safety margin to account for unobserved and hidden factors that may contribute to the WCET. MBTA can provide fully trustworthy WCET figures only when it can be guaranteed that all possible paths and execution conditions for a program-processor pair have been exhaustively observed. With respect to program paths, full coverage cannot be generally met within an industrial qualification process where timing testing typically reuses figures obtained during the functional verification phase.

The Single-Path Approach [26] sloughs off the complexity of path coverage by potentially reducing all programs to a single execution path. To this extent, specific compiler support and special processors supporting constant-time predicated instructions [27] are advocated. This however poses some requirements on the platform and compiler infrastructure, which might break consolidated industrial practice. On the opposite side of the spectrum of potential solutions, methods based on genetic algorithms and model checking have been proposed, with alternate fortune, in the last years to automatically generate the input vectors required to achieve full path coverage [28], [29]. However, the genetic algorithm these methods rely on are generally exposed to local minima/maxima that can prevent them from being successfully applied.

The role of input data in MBPTA was recognized recently [5], [7], [8]. The path representativeness problem was not exhaustively examined in [5] where the authors were relying on the user to be able to provide a set of relevant paths.

As we already observed, this assumption does not generally hold in practice as the user does not typically owe the tools or heuristics to capture relevant paths with respect to (p)WCET. The importance of providing representative inputs to MBPTA was first explicitly raised in [6] contextually with the proposal of Path Upper Bounding (PUB), a method to upper bound all possible paths in a program by artificially balancing the different branches of conditional control flow constructs. Although they may share their motivations, EPC and PUB differ in requirements: PUB is applied on a semantically-preserving extended version of the program, which may require an ad-hoc qualified compiler, while EPC [7] relies on the collection of additional information. The need for path representativeness was recently reinforced in [8], where analysis inputs were identified as a feature critically affecting MBPTA results. Some other works on MBPTA seem to disregard the path representativeness problem and focus exclusively on the statistical fitness of their model [30].

VI. Conclusions

MBPTA results are only valid for the execution conditions that have been captured at analysis time, and cannot be generalized to include unobserved paths. Exhaustive coverage by test is not a practical option for both the complexity and costs it implies. Measurement-based methods generally assume that the user is responsible for providing the input vectors that trigger the most relevant paths in the program. In practice, however, the user is not provided any means to collect or consolidate such knowledge.

The EPC approach, first introduced in [7], builds on the concept of probabilistic path independence to synthetically extend the path representativeness of MBPTA results to include also unobserved paths. In this work, we presented the challenges we faced and the solution we adopted in our attempt to bring EPC from an academic prototype to a solid toolchain. Despite its inherent complexity, EPC could be implemented on top of an industrial-quality toolchain and successfully evaluated against a real-world program. The initial results results on a realistic application are quite encouraging and allowed us to identify some room for improvement. As future work we plan to further consolidate the EPC implementation, in particular by improving the amount of contextual information to be considered in the analysis. We also plan to extend its evaluation against representative applications from other industrial domains, knowing that each of them has its own special traits and poses different challenges to novel technology concepts.

Acknowledgments

This work has received funding from the European Community's Seventh Framework Programme [FP7/2007-2013] under grant agreement 611085 (PROXIMA, www.proxima-project.eu). This work has also been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P and the HiPEAC Network of Excellence. Jaume Abella has been partially supported by the MINECO under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717. The authors are grateful to Antoine Colin from Rapita Ltd. for his precious support.

References

[1] O. Scheickl, C. Ainhauser, and P. Gliwa, "Tool support for seamless system development based on autosar timing extensions," in *Proceedings of Embedded Real-Time Software Congress (ERTS)*, 2012.
[2] S. Law and I. Bate, "Achieving appropriate test coverage for reliable measurement-based timing analysis," in *Euromicro Conference on Real-Time Systems, ECRTS*, 2016.
[3] 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.
[4] E. Mezzetti and T. Vardanega, "On the industrial fitness of WCET analysis," *11th International Workshop on Worst-Case Execution-Time Analysis*, 2011.
[5] L. Cucu-Grosjean *et al.*, "Measurement-based probabilistic timing analysis for multi-path programs," in *ECRTS*, 2012.
[6] L. Kosmidis *et al.*, "PUB: Path upper-bounding for measurement-based probabilistic timing analysis," in *ECRTS*, 2014.
[7] M. Ziccardi *et al.*, "EPC: Extended Path Coverage for Measurement-Based Probabilistic Timing Analysis," in *Real-Time Systems Symposium, 2015 IEEE*, 2015, pp. 338–349.
[8] G. Lima, D. Dias, and E. Barros, "Extreme value theory for estimating task execution time bounds: A careful look," in *Euromicro Conference on Real-Time Systems (ECRTS)*, 2016.
[9] S. Kotz and S. Nadarajah, *Extreme Value Distributions: Theory and Applications*. Imperial College Press, 2000.
[10] M. Paolieri *et al.*, "Hardware support for wcet analysis of hard real-time multicore systems," in *ISCA*, 2009.
[11] L. Kosmidis et al., "Probabilistic timing analysis and its impact on processor architecture," in *DSD*, 2014.
[12] F. Cazorla *et al.*, "PROARTIS: Probabilistically analysable real-time systems," *Transactions on Embedded Computing Systems*, 2013.
[13] L. Kosmidis et al., "A cache design for probabilistically analysable real-time systems," in *DATE*, 2013.
[14] E. Mezzetti et al., "Randomized caches can be pretty useful to hard real-time systems," *LITES*, vol. 2, no. 1, 2015.
[15] E. Mezzetti and T. Vardanega, "A rapid cache-aware procedure positioning optimization to favor incremental development," in *19th IEEE RTAS*, 2013.
[16] Special Committee of RTCA, "DO-178C, Software Considerations in Airborne Systems and Equipment Certification," 2011.
[17] L. Kosmidis et al., "Probabilistic timing analysis on conventional cache designs," in *DATE*, 2013.
[18] E. Diaz *et al.*, "Mitigating software-instrumentation cache effects in measurement-based timing analysis," in *Proceedings of the 16th International Workshop on Worst-Case Execution Time (WCET) Analysis*, 2016.
[19] C. Hernandez *et al.*, "Random modulo: A new processor cache design for real-time critical systems," in *Proceedings of the 53rd Annual Design Automation Conference*, ser. DAC '16, 2016.
[20] "Nexus 5001 forum." [Online]. Available: http://www.nexus5001.org
[21] "ARM® CoreSight©ip." [Online]. Available: https://www.arm.com/products/system-ip/coresight-debug-trace
[22] B. Dreyer *et al.*, "Precise Continuous Non-Intrusive Measurement-Based Execution Time Estimation," in *15th International Workshop on Worst-Case Execution Time Analysis (WCET 2015)*, 2015.
[23] B. Dreyer *et al.*, "Continuous Non-Intrusive Hybrid WCET Estimation Using Waypoint Graphs," in *16th International Workshop on Worst-Case Execution Time Analysis (WCET 2016)*, 2016.
[24] R. Kirner and P. Puschner, "Classification of Code Annotations and Discussion of Compiler Support for Worst-Case Execution Time Analysis," in *5th International Workshop on Worst-Case Execution Time Analysis*, 2005.
[25] J. Abella *et al.*, "On the comparison of deterministic and probabilistic wcet estimation techniques," in *ECRTS*, 2014.
[26] P. Puschner, "The single-path approach towards WCET-analysable software," in *International Conference on Industrial Technology*, 2003.
[27] S. A. Mahlke *et al.*, "A comparison of full and partial predicated execution support for ilp processors," in *ISCA*, 1995.
[28] I. Wenzel *et al.*, "Automatic timing model generation by cfg partitioning and model checking," in *DATE*, 2005.
[29] S. Bünte *et al.*, "Improving the confidence in measurement-based timing analysis," in *ISORC*, 2011.
[30] L. Santinelli *et al.*, "On the Sustainability of the Extreme Value Theory for WCET Estimation," in *14th International Workshop on Worst-Case Execution Time Analysis*, 2014.