A Framework For The Evaluation Of Measurement-based Timing Analyses

Benjamin Lesage
University of York
York, UK
bl778@york.ac.uk

David Griffin
University of York
York, UK
david.griffin@york.ac.uk

Frank Soboczenski
University of York
York, UK
frank.soboczenski@york.ac.uk

Iain Bate
University of York
York, UK
iain.bate@york.ac.uk

Robert I. Davis
University of York and
Inria Paris-Rocquencourt
rob.davis@york.ac.uk

ABSTRACT

A key issue with Worst-Case Execution Time (WCET) analyses is the evaluation of the tightness and soundness of the results produced. In the absence of a ground truth, i.e. the Actual WCET (AWCET), such evaluations rely on comparison between different estimates or observed values.

In this paper, we introduce a framework for the evaluation of measurement-based timing analyses. This framework uses abstract models of synthetic tasks to provide realistic execution time data as input to the analyses, while ensuring that a corresponding AWCET can be computed. The effectiveness of the framework is demonstrated by evaluating the impact of imperfect structural coverage on an existing measurement-based probabilistic timing analysis.

1. INTRODUCTION

The primary requirement for any WCET analysis is that the Computed WCET (CWCET) that it produces is sound, i.e. upper bounds the Actual WCET (AWCET). Determining the AWCET for realistically complex systems requires in the general case exhaustive testing [22], i.e. full state coverage. Demonstrating the soundness of an analysis in this context often relies on strong assumptions about the underlying system, such as the absence of timing anomalies [18]. A second requirement is the tightness of the CWCET, i.e. the difference between the CWCET and the AWCET. In the absence of knowledge regarding the AWCET, most contributions are restricted to comparisons with other CWCET obtained through different techniques.

The High Watermark (HWM), the highest observed execution time, is sometimes used as a substitute for the AWCET, as a baseline for comparison [19]. Lower CWCETs, closer the HWM, are assumed to be tighter estimates [13]. With the exception of simple systems, there is no guarantee that the HWM constitutes appropriate grounds for comparison. It is perfectly possible for a method, e.g. the original cache persistence analysis [21], to produce CWCET $C1$ lower than another, e.g. the revised analysis [13], if the former estimate $C1$ is optimistic, i.e. $HWM < C1 < AWCET$. Thus the absence of knowledge about the AWCET of a task impedes evaluation of the soundness and tightness of timing analyses. Measurement-based timing analyses require minimal knowledge of the program (task) and the hardware platform. Applied to a real system, they collect execution time measurements (samples) and require guarantees on the conditions under which these samples are collected, for example full path coverage and representative input data. Given these inputs, they determine a CWCET.

For the purposes of evaluating measurement-based timing analyses against a precisely defined ground truth (i.e. a known AWCET), one can safely abstract the platforms and tasks used in the evaluation of such methods provided that the samples fed into the analysis remain representative of realistic timing behaviours.

In this paper, we introduce a framework for the evaluation of measurement-based timing analyses. Instead of providing timing measurements from a real platform running real tasks for input into the analysis, this framework provides realistic data from synthetic (abstract) tasks. The data used to represent the execution times for the basic blocks of these synthetic tasks are sampled from a real system thus ensuring that they represent realistic timing behaviours. Through restrictions on the abstract model used, the complexity of AWCET computation for the synthetic tasks can be controlled, and thus an AWCET can be computed providing a ground truth against which the CWCET produced by the measurement-based analyses can be compared.

We note that the framework does not provide an AWCET for real tasks on a real platform, but rather an AWCET for synthetic tasks, that is correct with respect to the realistic timing data (samples) that the framework provides as inputs for the analysis. As a proof of concept, we instantiate the framework in a solution tailored to the requirements and assumptions of an existing Measurement-Based Probabilistic Timing Analysis (MBPTA) approach [8]. The contributions of this paper are to:

- show that using this framework we can determine the ground truth (AWCET and the set of execution time profiles) for synthetic tasks;
- show that the realism of the generated measurements

RTNS Lille, France

is not adversely affected by our assumptions;

- show how the framework enables controlled experiments that evaluate the impact on an existing approach [8] of unsatisfied path coverage assumptions.

The structure of the paper is as follows. We first introduce a model of the execution time of a program in Section 2. Section 3 instantiates that model and presents a MBPTA-tailored framework to build programs, their AWCET and execution times. We discuss and evaluate the representativity of this approach in Section 4. Section 4 also applies this framework in a controlled experiment aimed at the validation of an existing WCET analysis. Section 5 positions this contribution in relation to related work. Finally, Section 6 concludes with a summary and discussion of future work.

2. TEMPORAL EXECUTION MODEL

This section defines a model of the execution of a program p on a platform h. Although the model can capture the definition of concrete platforms, it is intended to reason about simplified abstract platforms supporting tractable AWCET computation. Section 3 instantiates and populates such an abstract platform tailored to the evaluation of MBPTA.

A program is characterised by a finite set of paths through its basic blocks. Each basic block is a sequence of instruction with a single entry and exit, located respectively at the beginning and end of the block. In other words, execution of a basic block proceeds in order through all its instructions. A path is itself a finite sequence of instructions. We denote a program p using its paths Π and basic blocks B, $p = \langle \Pi, B \rangle$.

The behaviour of a run through program p depends on the initial state of the system and the executed basic blocks, captured by a path in p. A platform is hence first characterised by a finite set of possible states S exercised during the execution of tasks. The outcomes of a basic block, in terms of execution times and output states, also depend on the platform and the included features. We denote by $\kappa(b, s)$ the function that captures the outcomes of basic block b from input state s. We denote a platform h using its states S and outcome function κ, $h = \langle S, \kappa \rangle$.

The finiteness of S, Π and each path guarantees the termination of any program p. Without these conditions, timing analyses for the modelled platforms would be required to solve the halting machine problem. The definition of a state depends on the abstract platform and does not preclude complex models. As an example, a multi-core architecture could be represented provided the definition of state is complex enough to capture the whole state of the processor, and that of co-runners of the considered task.

Let us first define the execution time of a path on a deterministic platform. On such platforms, the execution of a basic block from a given input has a fixed execution time and output state, formally: $\kappa(b, s) \in (\mathbb{N}^0 \times S)$. Intuitively, the execution time of a path π from input state s is the sum of the contribution of the traversed blocks b_i and states s_i:

$$ET(\pi, s, h) = \sum_{1 \leq i \leq len(\pi)} t_i \quad (1)$$

$$s_0 = s \quad (2)$$

$$(t_i, s_i) = \kappa(b_i, s_{i-1}) \quad (3)$$

s_i captures the series of states traversed during the execution of path π. s_i is the output of the i^{th} block b_i in the path.

The worst-case execution time of a program p on a given platform h, is defined as its longest execution time across all possible runs, the valid combinations of paths and input states:

$$WCET(p, h) = max_{\pi \in \Pi, s \in S}(ET(\pi, s, h)) \quad (4)$$

2.1 Probabilistic execution time

Using a non-deterministic platform, e.g. including a random replacement cache, introduces variability in the execution time of each basic block and, as a consequence, that of the traversing paths. From the same input, a path does not necessarily result in a single execution time but a distribution of execution times. The temporal behaviour of a path is therefore best represented by an Execution Time Profile (ETP), a probability mass function (PMF) which attaches to each possible execution time its probability of occurrence. We now extend the notion of execution time of a path from a deterministic value, $ET(\pi, s, h)$, to a probability distribution $pET(\pi, s, h)$.

We need to consider the impact of a non-deterministic platform at the basic block level. The execution of a block b from input s may result in multiple outcomes, combinations of execution latencies and output states. We modify the outcome function $\kappa(b, s)$ to model the probability of a basic block to have a specific output and cost, formally: $\kappa(b, s) \in PMF(\mathbb{N}^0 \times S)$.

κ captures potential correlations between the resulting state and execution time. Consider as an example a single cache with an evict-on-access randomised replacement policy [9]. We access memory block m present in the input cache. The access to m first results in the eviction of a block, either m itself or another block. In the former case, m is loaded back in the same cache line at the cost of a miss and the output cache state is unchanged. The latter cases, the access suffers a hit latency but a block has been evicted. The access to m can only result in a hit latency if a block other than m is evicted.

To formally define $pET(\pi, s, h)$, we need to consider all possible outcomes of the first block b_1 in the path and their impact on the following blocks of the path π_1. The weight $\mathcal{P}'$ of an outcome (t', s') is its occurrence probability:

$$pET(\pi, s, h) = \sum_{(t', s', \mathcal{P}') \in \kappa(b_1, s)} \mathcal{P}' \times (t' + pET(\pi_1, s', h)) \quad (5)$$

Where b_1 is the first block in path π and π_1 is the remainder of the path after the execution of basic block b_1. Assume $\mathcal{D}$ is an execution time distribution. $t' + \mathcal{D}$ adds t', the contribution of b_1, to distribution $\mathcal{D}$. The likelihood to get execution time t becomes $(t' + \mathcal{D})(t) = \mathcal{D}(t - t')$. $\mathcal{P}' \times \mathcal{D}$ weights distribution $\mathcal{D}$ by probability $\mathcal{P}'$, $(\mathcal{P}' \times \mathcal{D})(t) = \mathcal{P}' \times \mathcal{D}(t)$. The sum of distributions, e.g. across outcomes of b_1, sums up the probability of t in all distributions, $(\mathcal{D}' + \mathcal{D})(t) = \mathcal{D}(t) + \mathcal{D}'(t)$.

The probabilistic worst-case execution time (pWCET) of a program p, similarly to the deterministic WCET, must upper-bound the execution time distribution of p across all valid combinations of paths and input states. Figure 1 illustrates this relation using the exceedance probability (1-CDF, $\mathcal{P}[\mathcal{D} \geq t]$) of two different paths and a valid, but not the tightest, pWCET. A distribution $\mathcal{D}$ upper-bounds another $\mathcal{D}'$ if the probability to exceed any execution time t is higher in $\mathcal{D}$ than in $\mathcal{D}'$ [15]. In other words, $\mathcal{D}$ is more pessimistic than $\mathcal{D}'$, $\mathcal{D} \geq \mathcal{D}' \Leftrightarrow \forall t, \mathcal{P}[\mathcal{D} \geq t] \geq \mathcal{P}[\mathcal{D}' \geq t]$.

The actual pWCET of a program is the smallest upper-bound across the ETP of all its runs. The notion of envelope $\sqcup$ of distributions captures this smallest upper-bound [14]. The envelope $\mathcal{X} \sqcup \mathcal{Y}$ of distributions $\mathcal{X}$ and $\mathcal{Y}$ is the smallest distribution greater than both $\mathcal{X}$ and $\mathcal{Y}$, $\forall \mathcal{Z}, \mathcal{Z} \geq \mathcal{X} \wedge \mathcal{Z} \geq \mathcal{Y} \Rightarrow \mathcal{Z} \geq \mathcal{X} \sqcup \mathcal{Y}$. Hence, we formally define $pWCET(p,h)$:

$$pWCET(p,h) = \bigsqcup_{\pi \in \Pi, s \in S} (pET(\pi, s, h)) \tag{6}$$

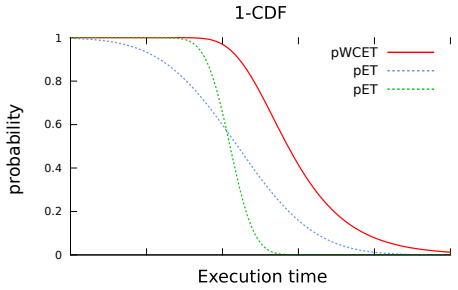

Figure 1: Relation between execution time of paths and a valid program pWCET.

2.2 pET and deterministic platforms

Given the notion of probabilistic execution time, we now consider how it applies in the context of deterministic architectures. As defined earlier, on a deterministic architecture $h^d = \langle S, \kappa^d \rangle$, the execution of a basic block b from a given input s produces a single outcome, $\kappa^d(b,s) = \{(t,s,1)\}$. This is a more restricted, simpler model than the non-deterministic one and the probabilistic execution time of a path (5) on a deterministic platform simplifies to its execution time (1):

$$\begin{aligned} pET(\pi, s, h^d) &= \sum_{(t',s',\mathcal{P}') \in \kappa^d(b_1,s)} \mathcal{P}' \times (t' + pET(\pi_1, s', h^d)) \\ &= \sum_{(t',s',\mathcal{P}') \in \{(t_1,s_1,1)\}} \mathcal{P}' \times (t' + pET(\pi_1, s', h^d)) \\ &= t_1 + pET(\pi_1, s_1, h^d) \\ pET(\pi, s, h^d) &= ET(\pi, s, h^d) \end{aligned} \tag{7}$$

3. SYNTHETIC TASK MODEL

We now present an experimental framework tailored to the evaluation of a number of important claims, notably the impact bias in execution time samples fed to measurement-based probabilistic timing analyses (MBPTA). Depending on the assessment and analysis method under evaluation, the considered model should obey different properties. In this scenario, MBPTA is a black box analysis and, provided sources of variability are either upper-bounded or randomised on the target platform, predicts the pWCET for the paths exercised by the samples fed to the analysis. This evaluation requires a baseline for comparison, an AWCET, to evaluate the soundness and the tightness of the method under normal and biased conditions. Computing the exact pWCET of a task is in the general case intractable. Instead, we rely on the controlled generation of synthetic tasks.

Our approach is outlined in Figure 2. We generate Abstract Syntax Trees (AST) from a parametrised program model, i.e. a definition of the possible paths between blocks. The combination of a temporal model and observations on a concrete platform defines an abstract platform, $\widetilde{h} = \langle \widetilde{S}, \widetilde{\kappa} \rangle$,

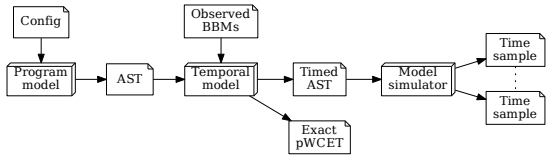

Figure 2: Illustration of the different steps involved in synthetic task and sample generation.

one that allows for an assessment of the robustness of the method. Temporal information, based on observations on a 'real' program, is then attached to the AST to produce a synthetic task for which the pWCET can be computed. The measurement-based analysis then uses samples built from controlled simulations of the synthetic tasks. We employ simplification of the program and temporal model to bring down the complexity of exact pWCET computation methods to a tractable level. Complexity is further controlled in the task synthesis process by parameters which limit the size of the set of paths in the AST.

In the following, we first introduce the independent block temporal model (Section 3.1) and the process used to collect observations at the block level (Section 3.2). Section 3.3 presents the AST generation step. The simulation of a synthetic task is discussed in Section 3.4. Finally, Section 3.5 describes the rationale behind efficient pWCET computation for synthetic tasks.

3.1 Independent block model

The temporal model of a platform captures the impact of a block as a function of its input state (see Section 2). The definition of the platform state is hence tightly coupled to the complexity of a model and impacts the difficulty of computing tight pWCET estimates. The construction of a temporal model can be approached in many ways, from an arbitrary execution model to exert properties of interest or using an abstraction of the features in a platform. Alternatively, observed behaviours can be matched against a simplified state for the observed platform. The latter case builds upon realistic data but requires the availability on the platform of information beyond timings. As an example, the state can discriminate timings based on the reuse distance of memory blocks, i.e. the distance between two accesses to the same block. As observed in static probabilistic timing analyses [1], the reuse distance is an important factor, although not the only one, in the contribution of caches to the execution time of basic blocks.

We first consider a simple model where each basic block b is assumed to have a single Execution Time Profile (ETP_b) giving the PMF for the execution time of the block. This effectively abstracts the effect of prior history and input state on the temporal contribution of a basic block. The behaviour of a block is independent of the behaviour of other or prior blocks in the program. The result is a stateless abstract platform $\widetilde{h} = \langle \widetilde{S}, \widetilde{\kappa} \rangle$:

$$\widetilde{S} = \emptyset, \quad \widetilde{\kappa}(b,s) = \widetilde{\kappa}(b) \tag{8}$$

A basic block having no impact on the state of the system, the output and temporal contributions of a basic block are uncorrelated. Therefore, the outcome of the execution of b only depends on its ETP_b:

$$\widetilde{\kappa}(b) = \{(t, \emptyset, \mathcal{P}) | (t, \mathcal{P}) \in ETP_b\} \tag{9}$$

MBPTA approaches advocate the use of architectures where

the temporal contribution of sources of variability are either upper-bounded or randomised during analysis. As demonstrated for random-replacement caches [9], these do not guarantee the ETP observed at the access or basic block level are independent, i.e. unaffected by prior accesses. As a first step in this study, we remove observed dependencies and show the representativity of the model is not impacted. The resulting model is an ideal case for the application of MBPTA approaches; the ETP of a path is effectively a combination of random events, modelled by the ETP of its basic blocks.

Note that the model exhibits properties inherent to the temporal contribution of tasks. The selected path and the ETP of its composing blocks contribute to the temporal distribution and variability of paths. Each potentially using widely different distributions. We evaluate the impact on accuracy of the independent block model in Section 4.

3.2 Gathering Basic Block Measurements

The proposed model relies on the availability for each basic block b of a matching ETP_b. To build such an ETP, we collect representative data, Basic Block Measurements (BBM), at the block level on a concrete application. Our prototypes use the Valgrind instrumentation framework [1] and the *FFmpeg* [2] application. The process first builds a control flow graph (CFG) of the instrumented application. The Cachegrind tool in Valgrind simulates the behaviour of the cache, hits and misses, and collects profiles of each traversed block. The collected profiles are then converted into execution time distributions to be used in the synthetic tasks. The different steps are illustrated in Figure 3. Successive iterations of the selected *FFmpeg* primitive, one per frame in the input feed, guarantees the collection of a large number of profiles under different inputs.

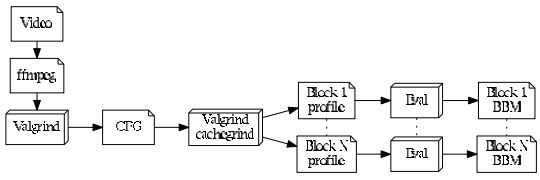

Figure 3: Illustration of the different steps involved in Valgrind-based sample collection.

Valgrind is a dynamic instrumentation framework tailored to support the development of analysis tools. Compared to more full-fledged simulators, the Valgrind solution has little deployment overhead. It supports instrumentation of any binary and input running on the host machine, including full system support. This eases and quickens the generation of numerous test samples, allowing for a large collection of BBM to synthesise tasks. Default tools further provide instrumentation focused on the collection of cache hit/miss profiles instead of costly full processor simulation. This eases understanding the sources of variability in the BBM.

FFmpeg is a multimedia transcoding library and includes a command line implementation. We instrumented the latter, focusing on the video decoding procedure. *FFmpeg* is one of the pieces of currently available software for which it is possible to assess the impact of using different input test vectors on the software being analysed. Indeed, each frame constitutes an input, and each video file is an opportunity to gather 'real' deployment data of the application with a different input vector. The widespread availability of video files allows for massive amounts of data. In contrast, popular real-time benchmark suites, e.g. TACLeBench [3] or Mälardalen [4], tend to rely on a fixed set of inputs, offering no variation.

Instrumentation collects the memory addresses accessed by successive calls to the *h.264* video frame decoding primitive and simulates their cache behaviour through the Cachegrind tool. The resulting cache hits and misses across the memory hierarchy are used to estimate the temporal behaviour of each call to the function. We modified Cachegrind to support the evict-on-miss random replacement policy [9] and to construct of the CFG of the instrumented program.

The caches and traversed execution paths are the main sources of execution time variability. The input of the *FFmpeg* application impacts the path followed during the execution of the task, and as such the behaviour of subsequent basic blocks. Given a basic block, the observed BBM is assumed to capture that block's temporal behaviour and the transposition of a BBM to an ETP_b is direct. This approach supports the independent block model by providing realistic ETPs, abstracting itself from the concrete effects of the history of execution on the cache. This in turns allows for a simple computation of the exact pWCET of a program. The BBM may in practice only capture an estimate of the execution time distribution of the corresponding basic block but the realism of the captured low-level distributions is not affected (see Section 4).

3.3 Program synthesis

As per our execution model, a program is represented by a set of paths Π, where each path is a finite sequence of basic blocks. Although it would be possible to randomly generate a number of arbitrary sequences of basic blocks, this is unlikely to result in representative program constructs. Furthermore, the size of the program representation would increase with the number of paths. We instead rely on a compact representation using AST. This section presents the considered structure and generation process.

Our prototype program generator synthesises AST from random combinations of `sequence`, `for`, `conditional`, and `block` nodes. Each type of node represents a common syntactic construct and possible paths between its blocks. AST further provide for a simplified reasoning on the worst-case execution time of a node as opposed to more free form representations such as control flow graphs which accept arbitrary transitions between basic blocks. The lack of expressiveness of AST, in terms of possible constructs or infeasible paths, can be circumvented by introducing new types of nodes, or duplicating part of the sub-trees, e.g. to remove a now infeasible path. This comes at an increase in the complexity of the structure.

Starting from the root of the tree, we randomly select a node type and create the appropriate node. The generator repeats the process for each of the children of the newly created node, selecting a node type, creating the node and its children. To control the size of the generated trees, the maximum width and depth of the tree are parametrised and enforced through restrictions on the maximum autho-

[1] http://www.valgrind.org
[2] http://ffmpeg.org
[3] http://www.tacle.knossosnet.gr/activities/taclebench
[4] http://www.mrtc.mdh.se/projects/wcet/benchmarks.html

rised nesting and maximum width of `sequence` and `conditional` nodes. Loop bounds are also randomly selected within parametrised ranges.

Figure 4 (and the corresponding Algorithm 1) illustrates the resulting structure. Semantically, conditions are evaluated in order (from left to right, `b{10}` then `b{11}` in `switch{9}`) until either one is true and the corresponding branch is executed, or the default one, if any, is executed (e.g. `b{7}` is the default branch in `ifelse{5}`). Sequences, such as `switch{9}; ifthen{3}`, similarly execute their children in order. The loop `for{12}` iterates exactly 19 times, over the sequence `b{13}; b{14}` in Figure 4. The loop condition `b{13}` is executed again at the end of the loop.

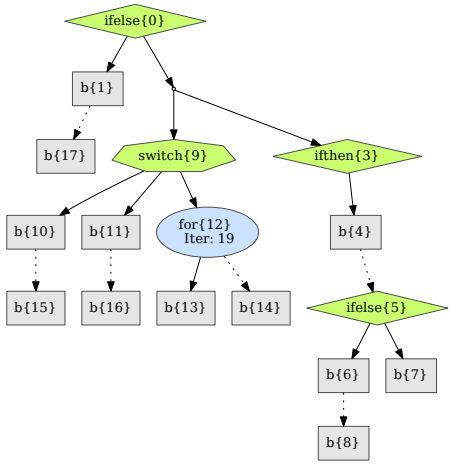

Figure 4: Example of a randomly generated AST including nested conditionals and a loop construct.

Algorithm 1 Pseudo code for Figure 4 AST.

```
/* ifelse{0} */
if b{1} then
    b{17}
else
    /* switch{9} */
    if b{10} then
        b{15}
    else if b{11} then
        b{16}
    else
        for i ∈ 1..19 /* b{13} */ do
            b{14}
    /* ifthen{3} */
    if b{4} then
        /* ifelse{5} */
        if b{6} then
            b{8}
        else
            b{7}
```

3.4 Simulation of a synthetic task

The simulation of a synthetic task enables the collection of execution time measurements. The resulting samples can be fed into measurement-based timing analyses. The execution of a task starts from its root node and progresses forward as per the semantic of the encountered node. The execution time of each encountered basic block b is picked at random according to its ETP_b. The sum of execution times picked across all traversed blocks provides an execution time for that run. Upon a `conditional` node, a decision is required to progress. The model does not enforce the selection of a specific branch, or a path. We defer the selection of the path executed during a run of the task to a sampling strategy. The sampling strategies can, as an example, allocate a selection probability to each path or define an ordering between paths executed in successive runs. The selected strategy depends on the claim under evaluation.

3.5 pWCET in the independent block model

An AST represents syntactic structures of a program as nodes in the tree. The execution time of a node is a simple function of the temporal behaviour of its children. This is the approach followed by tree-based (p)WCET computation approaches [17, 4]. The WCET of a task is represented as an expression of the WCET of its basic blocks, i.e. leaves in the tree. This expression captures the WCET of each node as the maximum of the execution time of all its paths. As an example, the WCET of a `conditional` node is that of its longest branch, including the execution time of conditions leading to the execution of that branch.

Tree-based WCET computation approaches rely on simple operations on execution times. However, this requires knowledge of an upper-bound of the temporal behaviour of each basic block. Block-level analyses can produce such execution times valid irrespective of the incoming path. The pessimism of the analysis introduced at the block level is reflected at the program level. In the independent block model, the ETP_b of a basic block naturally fits those constraints. It is a single execution time distribution valid irrespective of the paths followed to block b. The principles of tree-based WCET analyses can be applied to efficiently compute the actual pWCET of a program in the independent block model. We now show the ETP of different blocks can be added using the convolution operator ($\otimes$) and maximised by the envelope ($\sqcup$) one.

Let us first consider the computation of the probabilistic execution time of a path π. As described in the general case (5), it is a combination of the ETP of its composing blocks while propagating the state of the platform. The abstract platform in the independent block model is stateless. As a consequence, we can specialise the definition of probabilistic execution time in (5) to our abstract platform:

$$pET(\pi, s, \widetilde{h}) = \widetilde{pET}(\pi) \tag{10}$$

With b_1 denoting the first basic block in path π and π_1 the remainder of path π after the execution of b_1, we have:

$$\widetilde{pET}(\pi) = \left(\sum_{(t',\mathcal{P}') \in ETP_{b1}} \mathcal{P}' \times (t' + \widetilde{pET}(\pi_1)) \right)$$

The likelihood $\widetilde{pET}(\pi)(t)$ that path π has an execution time t is a sum of weighted distributions over ETP_{b1}. From the combinations rules in Section 2, it can be simplified as:

$$\begin{aligned} \widetilde{pET}(\pi)(t) &= \sum_{(t',\mathcal{P}') \in ETP_{b1}} \mathcal{P}' \times (t' + \widetilde{pET}(\pi_1))(t) \\ &= \sum_{(t',\mathcal{P}') \in ETP_{b1}} \mathcal{P}' \times \widetilde{pET}(\pi_1)(t - t') \\ \widetilde{pET}(\pi)(t) &= \sum_{(t',\mathcal{P}') \in ETP_{b1}} ETP_{b1}(t') \times \widetilde{pET}(\pi_1)(t - t') \end{aligned} \tag{11}$$

Obviously, if a basic block b_1 (respectively path π_1) cannot result in an execution time of t' (respectively $t-t'$), it has an occurrence probability of 0 in ETP_{b1} ($\widetilde{pET}(\pi')$). We can indifferently define $\widetilde{pET}(\pi)(t)$ over all possible execution times or only those in ETP_{b1}:

$$\begin{aligned}\widetilde{pET}(\pi)(t) &= \sum_{t'=-\infty}^{+\infty} ETP_{b1}(t') \times \widetilde{pET}(\pi_1)(t-t') \\ &= (ETP_{b1} \otimes \widetilde{pET}(\pi_1))(t)\end{aligned} \tag{12}$$

In other words, the probabilistic execution time of any path π is the convolution ($\otimes$) of the ETPs of its composing blocks: $\widetilde{pET}(\pi) = \bigotimes_{0<i\leq len(\pi)} ETP_{b_i}$

The convolution of two execution time distributions $\mathcal{X}$ and $\mathcal{Y}$ adds weighted execution times from a distribution to the other (12). Execution times are by definition positive or null values. Hence, the likelihood to exceed an execution time t is lower in execution time distribution $\mathcal{X}$ alone than in its convolution with execution time distribution $\mathcal{Y}$, $\mathcal{P}[\mathcal{X} \otimes \mathcal{Y} \geq t] \geq \mathcal{P}[\mathcal{X} \geq t]$. The convolved distribution is greater than the original, $(\mathcal{X} \otimes \mathcal{Y}) \geq \mathcal{X}$. This is also intuitively proved by induction, from the smallest execution time t in $\mathcal{X}$ (or $\mathcal{Y}$). Another consequence is the distributivity of the envelope $\sqcup$ and convolution $\otimes$ operators, a relation similar to that of the max and $+$ operators on positive integers:

$$(\mathcal{D} \otimes \mathcal{X}) \sqcup (\mathcal{D} \otimes \mathcal{Y}) = \mathcal{D} \otimes (\mathcal{X} \sqcup \mathcal{Y}) \tag{13}$$

pWCET computation at the node level

The pWCET of a program is the envelope of the execution time distributions of all possible outgoing paths (6). To each type of node n is attached a different semantic; the type of a node n defines the set of paths Π_n it generates from its children. We now define the pWCET of a node n in our model, $\widetilde{pWCET}(n)$. The pWCET of a program p is then defined as the pWCET of its root:

$$\begin{aligned}pWCET(p) &= \widetilde{pWCET}(root(p)) \\ &= \bigsqcup_{\pi \in \Pi_{root(p)}} (\widetilde{pET}(\pi))\end{aligned} \tag{14}$$

If n is a `block`, there is only one single path in Π_n, through basic block n, $\Pi_n = \{(n)\}$. The temporal behaviour of b is captured by the platform in the matching ETP_n:

$$\widetilde{pWCET}(n) = ETP_n \tag{15}$$

If n is a `sequence` of nodes, $n_1; n_2; ...; n_K$, its children are executed in order. The paths followed in each node are independent. The execution of a specific path in child n_i has no impact on the behaviour of node n_{i+1}. All combinations between paths of nodes n_1 to n_K must be considered, $\Pi_n = \Pi_{n1} \times ... \times \Pi_{nK}$:

$$\begin{aligned}\widetilde{pWCET}(n) &= \bigsqcup_{\pi \in \Pi_n} \widetilde{pET}(\pi) \\ &= \bigsqcup_{(\pi_1,...,\pi_K) \in \Pi_n} \left(\widetilde{pET}(\pi_1) \otimes ... \otimes \widetilde{pET}(\pi_K)\right)\end{aligned} \tag{16}$$

From (13) then (14), we have:

$$\begin{aligned}&\widetilde{pWCET}(n) \\ &= \left(\bigsqcup_{\pi_1 \in \Pi_{n1}} \widetilde{pET}(\pi_1)\right) \otimes ... \otimes \left(\bigsqcup_{\pi_K \in \Pi_{nK}} \widetilde{pET}(\pi_K)\right) \\ &= \widetilde{pWCET}(n_1) \otimes ... \otimes \widetilde{pWCET}(n_K)\end{aligned} \tag{17}$$

If n is a `conditional`, after the execution of a condition (e.g. `b{10}` in Algorithm 1), either the corresponding branch (`b{15}`) is executed or the next condition (`b{12}`) is evaluated. After the last condition fails, the default branch d is executed instead. We denote $\Pi^n_{c_i}$ the set of paths originating from condition c_i in `conditional` n, i.e. paths starting in c_i through the branch or the next condition. Π_{c_i} denotes the set of paths within condition c_i. r_i is the branch executed if c_i is verified, and d the default branch in the conditional:

$$\begin{aligned}&\Pi_n = \Pi^n_{c_0} \\ &\Pi^n_{c_K} = \Pi_{c_K} \times (\Pi_{r_K} \cup \Pi_d) \\ &\Pi^n_{c_i} = (\Pi_{c_i} \times \Pi_{r_i}) \cup (\Pi_{c_i} \times \Pi^n_{c_{i+1}}) = \Pi_{c_i} \times (\Pi_{r_i} \cup \Pi^n_{c_{i+1}})\end{aligned} \tag{18}$$

Hence, the pWCET of `conditional` node n is:

$$\widetilde{pWCET}(n) = \widetilde{pWCET}(n_0) \tag{19}$$

$$\widetilde{pWCET}(n_K) = \widetilde{pWCET}(c_K) \otimes \left(\widetilde{pWCET}(r_K) \sqcup \widetilde{pWCET}(d)\right)$$

$$\begin{aligned}\widetilde{pWCET}(n_i) &= \bigsqcup_{\pi \in \Pi^n_{c_i}} \widetilde{pET}(\pi) \\ &= \widetilde{pWCET}(c_i) \otimes \left(\widetilde{pWCET}(r_i) \sqcup \widetilde{pWCET}(n_{i+1})\right)\end{aligned}$$

If n is a `loop`, its execution is a sequence of i iterations through the loop condition and body, respectively n_h and n_b. The set of paths captured by n is $\Pi_n = (\Pi_{nh} \times \Pi_{nb})^i \times \Pi_{nh}$, including the last execution of the loop condition n_h. From the definition (16) of the pWCET of `sequence`, we derive:

$$\begin{aligned}\widetilde{pWCET}(n) &= \bigsqcup_{\pi \in \Pi_n} \widetilde{pET}(\pi) \\ &= \widetilde{pWCET}(n_h) \otimes \widetilde{pWCET}(n_b) \otimes ... \otimes \widetilde{pWCET}(n_h)\end{aligned} \tag{20}$$

As the convolution operation is commutative, we can reorder the operations such that the pWCET of the condition and body of the loop only need to be computed once:

$$\widetilde{pWCET}(n) = (\widetilde{pWCET}(n_h))^{(i+1)} \otimes (\widetilde{pWCET}(n_b))^i \tag{21}$$

The power operation on distributions uses the convolution operator and efficient computation decomposes the power operation into the convolution and memoization of smaller exponents, i.e. caching the results of lengthy calculations so that they may be re-used when required again. $(i+1)$ on the loop condition refers to the i iterations of the loop and the final evaluation of the condition. The use of a less constrained semantic on loops, where the loop may iterate less than the loop bound, does not change this equation. Any additional iteration adds to the execution time of the loop and therefore contributes to its worst-case execution time.

4. EVALUATION

In this section, we evaluate the proposed approach and the independent block model. First, we investigate the realism of the independent block model, that is any precision

loss induced by abstracting the state of the observed platform from the temporal model and assuming completeness of the BBM. We then evaluate a state-of-the-art measurement-based probabilistic timing analysis [8] using programs generated for our abstract platform. For the sake of consistency, the two sets of experiments use the same experimental conditions detailed below.

The **cache hierarchy** in Cachegrind features two, 64KB, 2-way L1 cache, for data and instruction, and a 256KB, 8-way unified L2 cache. All caches use the same 64B line size. The evict-on-miss random replacement policy is assumed in all levels of the cache hierarchy. We collect hits and misses data for each cache. The conversion from hits and misses to timings assumes access latencies of 1, 10 and 100 cycles respectively for the L1, L2 caches and main memory. This approach factors in the number of instructions in a block as each instruction takes at least 1 cycle (if it hits in the instruction L1 cache).

The **program synthesis** is also constrained. Conditions in `loop` and `conditional` nodes are restricted to simple basic blocks. The allocation of the BBM for an observed basic block to the ETP of a synthetic one, only pairs observed and synthetic blocks with similar structural constraints. In particular, the generator distinguishes blocks within loops, blocks within simple loops whose body comprises a single path, and dominators of the AST exit, whose execution is guaranteed in each run. Additional generation parameters, described in Table 1, ensure the generation of small applications with varied structures. Note that once the desired nesting level is reached, only basic blocks can be generated.

Table 1: Parameters from program synthesis.

`block`	
Weight	20
`sequence`	
Weight	5
Max. width	4
`conditional`	
Weight 1 condition, with default	5
Weight N conditions	1
Max. width	4
`loop`	
Weight	11
Iter range	(2, 16)
Max. Nesting	3

The complexity of the convolution ($\otimes$) and envelope ($\sqcup$) operations is tied to the size of the manipulated distributions. Given distributions $\mathcal{X}$ and $\mathcal{Y}$ each with N elements, the complexity of the general convolution operation, $\mathcal{X} \otimes \mathcal{Y}$, is $O(N^2)$ and $\mathcal{X} \otimes \mathcal{Y}$ may hold up to N^2 entries. As the computation of the pWCET of a program progresses, the size of the manipulated distribution increases. Although more efficient implementation of the discrete convolution operation exist, e.g. FFT-based discrete circular convolutions [6], they rely on padding the input distributions, thus increasing the memory requirements of the algorithm for widespread distributions. Efficient sparse convolution algorithms [2] may require knowledge on the size of input distribution.

We rely on **Lossy compression** [12], as proposed in the context of static probabilistic timing analyses, to reduce the cost of the computation of AWCET of synthetic programs. Execution times whose occurrence probability falls beneath a selected threshold (10^{-17}), matching the desired exceedance probability for the task, are soundly removed. Their probability is accumulated in the largest value in the distribution. Compression results in a slight loss in tightness, but improves the computation time of the pWCET analysis. Similarly, we use resampling [16] to bound the complexity of the envelope and convolution operations (up to 16000 entries per distribution). Resampling merges the occurrence probability of consecutive entries in the distribution into the largest one.

The **input vectors** to the *FFmpeg* application are listed in Table 2. All video files were obtained from the Internet Archive movies collection. To ensure a similar frame size in all cases, our observations used the 512KB MPEG4 encoding of each source file. We only collect BBM over the first 8000 frames of each movie.

Table 2: Considered input vectors.

Id	Title
nosf	Nosferatu, eine Symphonie des Grauens (1922)
kung	Return of the Kung Fu dragon (1976)
plan	Plan 9 from Outer Space (1959)
phop	The Phantom of the Opera (1925)

4.1 Does the framework produces realistic execution time traces ?

In this section, we evaluate the realism of the proposed approach. In particular, we observe the impact of the state-independence and completeness assumptions on the observed BBM and paths. To that end, we collect end-to-end timing measurements of the frame decoding process on the *FFmpeg* application, alongside a list of traversed basic blocks for each observation. This provides an *observed* execution time distribution.

We match the observed distribution against a synthetic one. The synthetic distribution is constructed by simulating each of the observed paths using our model, that is execution times are generated as the sum of values randomly picked inside the ETP of the traversed blocks. The ETP of each block is built from all its occurrences across observed paths. The simulated and observed distributions are built in each case using the same input vector (identified by its id in Table 2). For the sake of brevity, we only present the 1-CDF for the *phop* and *plan* input vectors respectively in Figures 5 and 6. The framework behaves similarly for *phop* and *nosf* on one hand, and *plan* and *kung* on the other.

As illustrated in Figures 5 and 6, the synthetic execution time distribution, obtained through simulation in our model of the observed paths, exhibits a similar shape to the observed one. Variations due to changes in the input vector are reflected upon the exercised path and, as a consequence, both the observed and simulated distributions. The two distributions are very close. In the worst-case *phop*, the majority of simulated runs are within 10% of their observed counterparts. Conservative assumptions during the conversion of observed block-level cache profiles to ETPs, explain the increase in execution time of the simulations over the observations in Figure 5. This occurs when the number of memory operations within a basic block, as reported by the instrumentation framework, is not constant.

Variability on the observed platform is tied to both the observed path and the behaviour of the randomised cache hierarchy. The experiment traverses the same paths, i.e. sequence of basic blocks, in both simulations and observa-

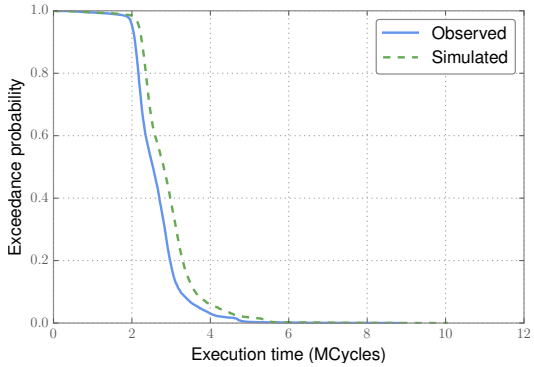

Figure 5: Simulated and observed execution time distributions for the *phop* input vector.

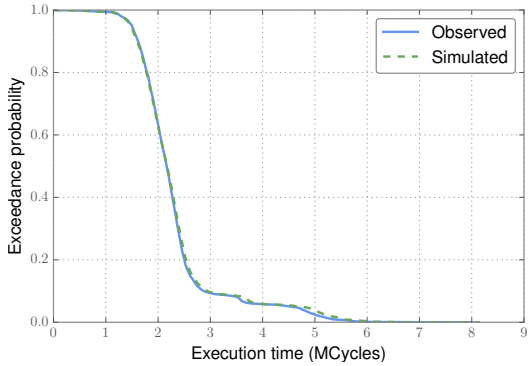

Figure 6: Simulated and observed execution time distributions for the *plan* input vector.

tions. The difference between the two distributions comes from differences in the execution time picked in each basic block. These results underline the adequacy of our assumptions, the realism of the BBM and the importance of the path in the variability of execution time beyond history of execution. The observed execution time at the block level in turn is indeed impacted by the prior sequence of accesses and the replacement policy. The use of the random policy implies that, barring guaranteed hits, each access has a miss probability impacted by prior hits and misses. Different sequences of addresses with similar probabilistic behaviours can therefore have similar impact on the execution of a block. As compared to deterministic policies, such as LRU, there is also less of a threshold effect where a single additional access in the history can turn a guaranteed hit into a miss.

4.2 Robustness of MBPTA

The following section presents a controlled experiment for the evaluation of timing analyses based on Extreme Value Theory (EVT), i.e. MBPTA using hardware and software customised for the purpose. In the original contribution [8], MBPTA was been described as producing valid pWCET estimates only for those paths exercised by the execution time samples fed to the method. To understand the importance of this requirement, we evaluate the robustness of the method in presence of path-related bias in the samples fed to the analysis. This is but one example of many claims that could be evaluated as part of a certification case for MBPTA [11].

To assess the soundness and precision of the analysis requires the knowledge of the AWCET of a task. The use of a simple temporal model, the independent block model, allows such a computation. Only results of interest are presented in the following, to illustrate general trends or special cases. The framework indeed allows for the generation of copious configurations of synthetic tasks along with the attached ground truth, an AWCET and controlled samples of execution times (see Section 3).

To build sets of synthetic execution time samples, a sample strategy must be considered. The execution time of a run is obtained by randomly picking execution times in the ETPs of the traversed blocks. Different strategies can provide different levels of control over the set of basic blocks, paths or execution times covered by a given sample. Our strategy decides the outcome of each encountered `conditional` node at random. `loop` always take the maximum number of iterations. All outcomes, branches in the condition, have the same weight. This effectively executes a random path, but allows for a good coverage of the basic blocks in a simple AST. To support the introduction of bias towards specific parts of the program, we introduce the notion of blacklisted nodes in the sampling strategy. The strategy only picks paths among the ones which traverse no blacklisted nodes.

We use a set of 100 randomly generated tasks. For each, we generate two samples of 8000 runs and apply MBPTA to predict the pWCET of the synthetic task. Only tasks with a sensible number of paths (less than 8000) are kept during the generation process, ensuring our baseline configuration can fulfil the requirements of MBPTA in terms of path coverage. Under these conditions, the sampling strategy consistently provides a good coverage of paths.

To introduce bias in the samples fed to the analysis, samples are collected with an increasing number of randomly blacklisted blocks. The process only selects those blocks that do not dominate the exit of the task, i.e. blocks whose execution is not mandatory in every run. Each sample is then fed to MBPTA to predict the pWCET of the task. To allow for a comparison across all tasks, we read off the predicted pWCET at exceedance probability 10^{-9} and normalise it over the exact value at the same probability.

Figure 7 presents the result of those experiments as a box and whisker plot. For each experiment, i.e. number of blocks ignored in a task, the box (dashed blue) captures the first (25%), third quartiles (75%) and median (red line, 50%) of the normalised predictions across the different samples. If sufficient data is available, the whiskers (black dash) represent the 9^{th} and 91^{st} percentiles. Orange diamonds mark outliers. We also consider in Figure 8 the random removal of nodes instead of leaves in the AST. Complete subtrees may as a result be ignored by the sampling process.

We first focus on cases where all paths are are captured by the analysed sampled (the leftmost entry of Figure 7 and 8). Both have been obtained under the same conditions and produce similar results in terms of soundness. When all paths are covered, only a marginal number of generated tasks were found unfit for analysis through MBPTA, failing the minimum run test.

The omission of random blocks, leaves within the AST, bears little effect on the behaviour of the analysis as illustrated in Figure 7. If the removal decreases the number of sampled paths, since the blacklisted blocks are non-mandatory the remaining paths are similar to the removed

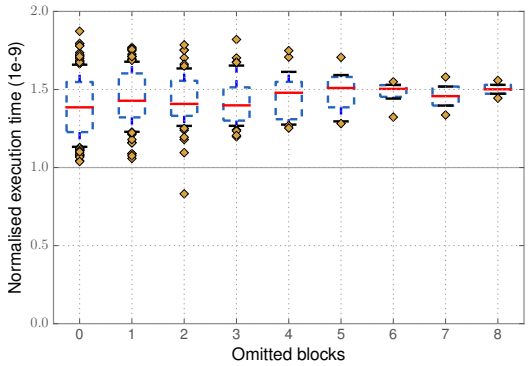

Figure 7: Precision at 10^{-9} of the MBPTA-predicted pWCET with an increasing number of blacklisted blocks.

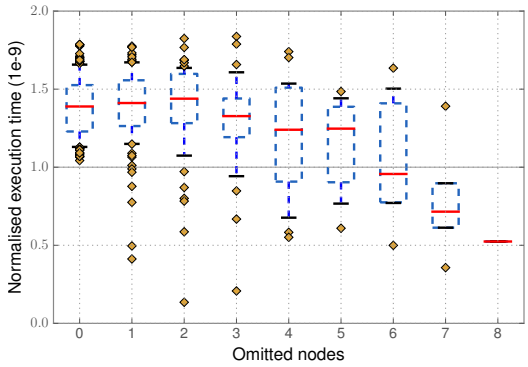

Figure 8: Precision at 10^{-9} of the MBPTA-predicted pWCET with an increasing number of blacklisted nodes.

ones traversing alternative blocks nested in the same structures. When the removal is more local, as is the case when nodes are omitted in Figure 8, the focus on a more consistent set of paths reduces the overall span of predicted values. However, the loss of program coverage also degrades the soundness of the predicted results. Note that in later stages some of the exposed variability could be argued to be the result of randomised task parameters as well. As an example, between 6 and 8 omitted nodes in Figure 7, less samples are available as few tasks offer enough candidates for omission without removing all feasible paths.

The path coverage requirement is in practice difficult to achieve, even using reasonably-sized samples we had to reject 3 out of 5 generated tasks to ensure the practicality of the requirement. In our experiments, when this requirement was satisfied, MBPTA produced sound estimates. There is a reasonable argument that a less restrictive, more refined condition can be constructed biased towards the coverage of specific paths, e.g. the worst-case inducing ones as in probabilistic static analyses [14], can be constructed. This may alleviate the need for full path coverage while still ensuring sound results. Coverage of the basic blocks in the program is not a strong requirement provided the omitted blocks are spread across the program, i.e. similar alternatives are covered.

5. RELATED WORK

Static WCET analysis techniques [10, 5, 22] rely on the definition of abstract models of the analysed platform to estimate its temporal behaviour. As an example, abstract interpretation-based cache analyses [10] capture an upper-bound of the cache state before every basic block in the application. This provides guarantees on the absence or presence of blocks in the cache. More precise approaches [5] rely on more costly models, e.g. to distinguish different contexts and timings for a same basic block. Existing computation techniques provide ground for the definition of various abstract platforms. Our framework is however ill-suited to the evaluation of static analyses. To evaluate an analysis using a given model, would require a more precise model to define an evaluation platform. If exact pWCET computation using the latter is tractable, it is a likely replacement for the original analysis.

Measurement-based WCET analyses [4, 8] on the other hand rely on measurements obtained from the target platform to derive WCET estimates. Some approaches rely on the collection of supplemental information, e.g. the structure of the observed program [4], to cover unobserved configurations. On the other hand, black box approaches [8] may rely on compliant hardware to ensure the observed platform upper-bounds the deployed one. MBPTA [8] relies on non-deterministic architectures. The behaviour of a task is then estimated through extrapolation of the tail of the observed execution time distribution. The quality of the collected samples is therefore of prime importance.

The expression of the execution time of a program as a transition system between states of the platform is a natural process. It allows the definition of the behaviour of a platform at a low, instruction-level granularity [20]. This model was notably used to lay down a classification of timing anomalies [18]. The transition system model also offers a generic solution for the computation of the WCET of a task, through a collecting semantics exploring all possible states [7].

6. CONCLUSIONS

The main contribution of this paper is the introduction of a framework for the evaluation of measurement-based timing analyses. Instead of providing timing measurements from a real platform running real tasks for input into the analysis, this framework provides realistic data from synthetic (abstract) tasks. This allows a corresponding AWCET to be computed, enabling the soundness and tightness of the timing analysis results to be evaluated.

We applied this approach defining a simple independent block model, where basic block measurements from a real system were used to populate realistic ETPs for the basic blocks in synthetic tasks. (Execution time variability under this model is mostly due to the exercised path). Our evaluation, targeted at platforms with caches with a random replacement policy, showed that the proposed model produces realistic outputs when compared to timing data for the original task running on the real platform. Further, we evaluated the soundness and tightness of MBPTA, exploring the effects of bias in terms of missing input data (omitted blocks and nodes in the task's AST).

Our experiments showed that (as expected) a lack of path coverage, due to unobserved nodes in the AST, deteriorates the soundness of the analysis. When applicable and with full path coverage, the results produced by MBPTA were found to be sound, and reasonably tight w.r.t. the ground

truth computed by the framework.

One avenue for future research is to investigate the coverage needed by MBPTA to ensure that sound results are obtained. Our evaluation mostly focuses on the impact of randomised bias on the soundness and tightness of the resulting temporal estimates. More controlled forms of bias, as an example towards the best or worst-case paths could be considered. Bias can also manifest as the occurrence of dependencies between runs of the task, e.g. if successive runs operate in the same operation mode. The sampling strategy used in our evaluation effectively randomises the executed path selection, different strategies could be implemented to introduce and evaluate the impact of different forms of dependencies between consecutive runs.

Future work should also consider the introduction of different temporal models, in particular including dependencies between blocks of a task. Copulas have been considered in previous work to represent such dependencies [3] but require knowledge of the execution time distributions of basic blocks. This information is available as part of the temporal model, e.g. from basic block measurements. More focused models could allow for the evaluation of MBPTA in the presence of timing anomalies, through control of their occurrences on an abstract platform. Some control should also help in ensuring the computation of the exact pWCET of the synthetic tasks remain tractable.

Acknowledgement

The authors would like to thank Liliana Cucu-Grosjean and Adriana Gogonel from Inria Paris-Rocquencourt for their help in the implementation of the MBPTA analysis. This work was partially funded by the EU FP7 Integrated Project PROXIMA (611085) and the INRIA International Chair program.

7. REFERENCES

[1] S. Altmeyer and R. I. Davis. On the Correctness, Optimality and Precision of Static Probabilistic Timing Analysis. In 17^{th} *Conference on Design, Automation and Test in Europe (DATE)*, 2014.

[2] A. Amir, O. Kapah, and E. Porat. Deterministic length reduction: Fast convolution in sparse data and applications. In *Combinatorial Pattern Matching*, volume 4580 of *Lecture Notes in Computer Science*. 2007.

[3] G. Bernat, A. Burns, and M. Newby. Probabilistic timing analysis: An approach using copulas. *Journal of Embedded Computing*, 1(2):179–194, 2005.

[4] G. Bernat, A. Colin, and S. Petters. WCET analysis of probabilistic hard real-time systems. In *23rd Real-Time Systems Symposium (RTSS)*, 2002.

[5] S. Chattopadhyay and A. Roychoudhury. Scalable and precise refinement of cache timing analysis via path-sensitive verification. *Real-Time Systems*, 49(4), 2013.

[6] M. I. Cîrnu. Circular Convolution And Fourier Discrete Transformation. *Journal of Information Systems and Operations Management*, 7(2):280–287, 2013.

[7] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th Symposium on Principles of Programming Languages (POPL)*, 1977.

[8] L. Cucu-Grosjean et al. Measurement-Based Probabilistic Timing Analysis for Multi-path Programs. In *24th Euromicro Conference on Real-Time Systems (ECRTS)*, 2012.

[9] R. I. Davis, L. Santinelli, S. Altmeyer, C. Maiza, and L. Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *25th Euromicro Conference on Real-Time Systems (ECRTS)*, 2013.

[10] C. Ferdinand and R. Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999.

[11] P. Graydon and I. Bate. Realistic safety cases for the timing of systems. *The Computer Journal*, 57(5), 2014.

[12] D. Griffin, B. Lesage, A. Burns, and R. Davis. Lossy compression for static probabilistic timing analysis of random replacement caches. In *22nd Conference on Real-Time Networks and Systems (RTNS)*, 2014.

[13] B. K. Huynh, L. Ju, and A. Roychoudhury. Scope-Aware Data Cache Analysis for WCET Estimation. In *17th Real-Time and Embedded Technology and Applications Symposium*, 2011.

[14] B. Lesage, D. Griffin, S. Altmeyer, and R. Davis. Static Probabilistic Timing Analysis for Multi-path Programs. In *Real-Time Systems Symposium (RTSS), to appear*, 2015.

[15] J. López, J. Díaz, J. Entrialgo, and D. García. Stochastic analysis of real-time systems under preemptive priority-driven scheduling. *Real-Time Systems*, 40(2), 2008.

[16] D. Maxim et al. Re-sampling for Statistical Timing Analysis of Real-time Systems. In *20th Conference on Real-Time and Network Systems (RTNS)*, 2012.

[17] P. Puschner and C. Koza. Calculating the maximum execution time of real-time programs. *Real-Time Systems*, 1(2), 1989.

[18] J. Reineke, B. Wachter, S. Thesing, R. Wilhelm, I. Polian, J. Eisinger, and B. Becker. A Definition and Classification of Timing Anomalies. In *6th Workshop on Worst-Case Execution Time Analysis (WCET)*, 2006.

[19] L. Santinelli, J. Morio, G. Dufour, and D. Jacquemart. On the Sustainability of the Extreme Value Theory for WCET Estimation. In *14th Workshop on Worst-Case Execution Time Analysis (WCET)*, 2014.

[20] T. Schuele and K. Schneider. Abstraction of Assembler Programs for Symbolic Worst Case Execution Time Analysis. In *41st Annual Design Automation Conference (DAC)*, 2004.

[21] H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and Precise WCET Prediction by Separated Cache and Path Analyses. *REAL-TIME SYSTEMS*, 18, 1999.

[22] R. Wilhelm et al. The Worst-case Execution-time Problem — Overview of Methods and Survey of Tools. *ACM Trans. Embed. Comput. Syst.*, 7(3), May 2008.