Abstract—Measurement-Based Probabilistic Timing Analysis (MBPTA) responds to the challenge of analysing the timing behaviour of real-time software running on hardware deploying high-performance features (e.g., data caches). MBPTA provides a WCET estimate that upper-bounds the execution time of all paths in the program, even when the user-provided input vectors do not exercise the worst-case path. This significantly reduces the requirements imposed on the user to apply MBPTA. For Mälardalen and EEMBC respectively, MBPTA provides WCET estimates 5% and 11% higher than the WCET estimates computed with MBPTA.

I. INTRODUCTION

The increasing demand for computing power in Safety-Critical Real-Time Embedded Systems pushes system designers towards adopting high-performance processors with performance-improving features such as cache memories. However, those features challenge classic timing analysis methods such as static timing analysis (STA) and measurement-based timing analysis (MBTA).

STA [1] builds accurate timing models of the program under analysis and the hardware where it is run. Constructing those timing models for increasingly complex hardware and software is time consuming, which in turn raises the cost for computing tight Worst-Case Execution Time (WCET) bounds. MBTA [1] is not directly impacted by the use of more complex hardware and software. However, deriving trustworthy WCET estimates becomes overly complex in the presence of hardware features such as cache memories, which are prone to abrupt variations [2] in the execution time of applications when small changes occur in application’s execution conditions such as the placement of objects in memory.

Measurement-Based Probabilistic Timing Analysis (MBPTA) [3], [4] has been proposed to derive trustworthy WCET estimates on complex hardware. MBPTA, and in general Probabilistic Timing Analysis (PTA) techniques [5], obtain probabilistic WCET (pWCET) estimates that can be exceeded with a user-defined probability. This probability can be arbitrarily low, e.g., $10^{-12}$ per hour, thus not exceeding the acceptable failure rates for the different safety levels described in functional safety standards such as ISO26262 [6] (automotive) and DO-178C [7] (avionics) among others.

MBPTA provides pWCET estimates that upper-bound the execution time of those paths exercised with the input vectors provided by the user\(^1\). Unfortunately, in general, the user cannot determine all those paths leading to the highest execution times and hence, may not provide input vectors that exercise them. Typically, timing analyses rely on the ability of the user to provide loop bounds and/or generate input vectors exercising the highest loop bounds, which is already a challenging task. However, conditional control-flow constructs (CFC) such as if-then and if-then-else, are not easy to bound. This is so because detailed knowledge about the processor state (e.g., cache state) is required to determine which path (i.e., sequence of branches) across different CFC impacts execution time the most, with the number of paths growing exponentially with the number of dynamic CFC\(^2\).

In this paper we propose \textit{MBPTA}, a path upper-bounding method that extends MBPTA such that the provided pWCET estimates upper-bound any path in the program, even when the input vectors do not exercise the worst-case path. To that end, \textit{MBPTA} generates an extended version of the program by adding instructions in the different branches of CFC so that the execution time of any path of the extended program upper-bounds the execution time of all paths in the original program. Interestingly, the extended program can be used only to generate the pWCET at analysis time, while at deployment time the original unmodified program can be used. We also introduce a variant of MBPTA that reduces the overhead of the original version for some specific CFCs.

Unlike MBPTA that requires full-path coverage to provide the pWCET for a program, MBPTA requires few input vectors, i.e. end-to-end traversals of the program. This requirement is much less than basic block coverage, which in fact is usually required for functional verification. For instance, modified condition/decision coverage (MC/DC) is required for the highest design assurance level (DAL), DAL A, in avionics. Hence, in the general case, MBPTA makes no input vector be required for timing verification of the system beyond those input vectors provided for functional verification. This significantly reduces the cost of applying the analysis technique.

\textit{MBPTA} builds upon the observation that caches deploying random placement and replacement [8], a.k.a time-randomised caches, required for MBPTA are much less history-dependent than conventional caches based on, for instance, modulo placement and least recently used (LRU) replacement. Cache history dependence on PTA-compliant hardware platforms is made probabilistic and the dependence between the memory address assigned to a particular piece of data and its assigned set in cache is broken. As a result, upper-bounding the effect of the different branches of CFC in the state of a time-randomised cache can be done with lower complexity than for a time-deterministic cache.

Our results show that MBPTA provides pWCET estimates that upper-bound the execution time of every path in the program, while increasing pWCET estimates only by 11% for EEMBC benchmarks and 5% for Mälardalen with respect to the pWCET estimates provided by MBPTA using user-

\(^1\)user’ refers to the system designer, integrator or whoever uses MBPTA.

\(^2\)Dynamic CFC stands for the sequence of CFC traversed during execution. For instance, if there is one static CFC in the code but it is executed $N$ times within a loop, it leads to $N$ dynamic CFC producing up to $2^N$ different execution paths.
MBPTA aims at providing \( p_{WCET} \) estimates that hold under the execution conditions that may occur during actual operation. While those conditions likely are not exactly identical to those captured by the observation runs made at analysis time, analysis time conditions must still reproduce or upper-bound the probabilities of the execution times that may occur during operation [10]. As MBPTA uses EVT and EVT has its own requirements, namely that execution times to be modelled with random independent and identically distributed variables [11][12], MBPTA inherits them. However, MBPTA adds its own requirements on top of EVT’s ones, which serve MBPTA to achieve its own goal [3][10]. The correct application of MBPTA requires properly identifying all the sources of execution time variation of the program under study. The nature of each source of variability has to be understood and properly captured at analysis time to guarantee that \( p_{WCET} \) estimates hold during operation. The existence of an ETP per dynamic instruction (instance of that program instruction) enables the use of MBPTA [9].

### A. MBPTA-compliant Hardware Designs

MBPTA-compliant hardware designs exhibit several characteristics [5] including: lack of timing anomalies [13][14]; time-randomised caches [8]; and resources with data-dependent latency executing at their highest latency during analysis time, e.g., by deploying the worst-case mode [15], a technique that makes resources serving any request on their worst-case execution time, even if the request ends ahead of time.

At processor core level, we assume a pipelined processor in which each instruction takes a fixed latency, similar to LEON 4 processor [16] as it has been shown to be analysable with MBPTA techniques [8]. At cache level, we use the time-randomised caches proposed in [8], which implement random placement and replacement policies. In the case of random replacement we use evict-on-miss (EoM) policy which, on the event of a miss, randomly selects a victim line and replaces its contents with the contents of the new access. Time-randomised caches [8]; and resources with data-dependent latency. Randomised caches [8]; and resources with data-dependent latency have been shown to provide comparable average performance to that of time-deterministic caches (e.g. deploying modulo placement and LRU replacement) [8], [17].

Given the sequence of accesses \(<A_i,B_1,B_2,...,B_k,A_j>\), where \(A_i\) and \(A_j\) access the same cache line, and \(B_i, i \in [1..k]\), access different cache lines among them that is also different from the line accessed by \(A_i\) and \(A_j\), the hit probability of \(A_j\) can be approximated as [8]:

\[
P_{hit,A_j} = \left( \frac{W-1}{W} \right) \sum_{i=1}^{k} \frac{P_{miss,B_i}}{S} \cdot \left( \frac{S-1}{S} \right)^{k-1}
\]

\(W\) is the number of ways and \(S\) the number of sets in the cache. \(k\) is the reuse distance of access \(A_j\), that is the number of cache accesses in between \(A_j\) and the previous access to the same cache line, i.e. \(A_i\).

In [18] authors propose an upper-bound to the hit probability of accesses for a \(N\)-entry fully-associative cache, hence...
not using random placement, deploying EoM replacement: 
\[ P_{hit_{Aj}} = \left( \frac{N - k}{N} \right)^k \] if \( k < N \) and 
\[ P_{hit_{Aj}} = 0 \] if \( k \geq N \).

It is worth noting that MBPTA requires no derivation of an upper-bound probability of miss for cache accesses. Unlike SPTA (the static PTA counter part of MBPTA), MBPTA does not derive pWCET estimates by deploying convolution, but it is based on observations. MBPTA requires that cache accesses have an associated probability of hit/miss. The formula in [8] (Equation 2) is an approximation to the actual probability of hit of an access to a random placement random replacement cache. Its purpose is providing an intuition on the fact that cache hits/misses have a probabilistic nature, which is enough for the applicability of MBPTA. Following the same philosophy, approximation formulas for the hit/miss probability are shown in [19] for multi-level random caches.

From formulas above, intuitively we see that the higher the number of intermediate accesses between \( A_i \) and \( A_j \), the higher the reuse distance of \( A_j \) and hence the lower its hit probability.

### B. Path coverage

One of the most challenging steps in applying MBPTA and, in fact, any measurement-based technique, is the generation of test data. The user must provide a range of input data to the program, designed to stress the program and produce worst-case behaviour. The choice of the data may affect the timing behaviour of the software, hence properly selecting test vectors ensures trustworthiness and efficiency of the overall analysis. It is also the case that common code coverage criteria from the domain of functional testing, including Random Testing, Basic Block Coverage, Condition/Decision Coverage and Modified Condition/Decision Coverage (MC/DC), can underestimate the WCET in the context of measurement-based techniques [20]. In particular, MBPTA depends on the input vectors provided by the user in terms of (1) loop bounds and (2) WCET-relevant execution paths. Providing input vectors with WCET-relevant loop bounds can be regarded as attainable for the user in general given that the highest values are the WCET-relevant ones. However, determining which execution paths – out of the combination of all branches of all dynamic CFC – are the ones leading to the WCET is, at best, very difficult. Further, generating input vectors exercising those paths can be regarded as unattainable in the general case.

The analysis of the timing behaviour of caches and its interaction with control-flow analysis have been deeply analysed for deterministic systems [21], [1]. Time-randomised caches remove any dependence on the location of objects (e.g. code, libraries, OS, heap, etc.) in memory and so, on the particular cache location in which they are located, which simplifies the analysis of cache addresses across different paths. Despite that, the dependence among different accesses is not completely removed, but that dependence is made probabilistically modelisable. In Section III, we present how PUB builds upon the features of time-randomised caches to allow upper-bounding different branches of CFC at low cost.

### III. PRINCIPLES OF PUB

PUB allows MBPTA to derive pWCET estimates that probabilistically upper-bound the execution time of any path in the program even when the user-provided input vectors do not exercise the worst-case path. The picture on the left in Figure 2 shows the MBPTA flow, where end-to-end execution time measurements are collected for paths exercised with the user-provided input vectors. pWCET estimates obtained are only guaranteed to upper-bound the execution times of exercised paths. Nothing can be stated about any other execution path. Instead PUB, see right picture in Figure 2, operates either on the original source code or the object code adding instructions in the different branches of CFC such that, regardless of which particular branch is traversed in each dynamic CFC in the modified code, the pET of the extended program upper-bounds the pET of the original program for any traversal (i.e. path) of the conditional constructs. That is, if the original program has \( E^{org} \) different execution paths and the extended has \( E^{ext} \), the following equation holds, where \( pET(p) \) is the pET of the execution path \( p \):

\[ \forall (ep_i \in E^{ext}, ep_j \in E^{org}) : pET(ep_i) \geq pET(ep_j) \] (3)

Let us assume a CFC with two conditional branches, being the sequence of instructions executed in each branch \( IS^{left} \) and \( IS^{right} \) respectively. The main idea behind PUB is adding new operations \( O \), both core operations (e.g. add) and memory operations, with neutral impact on functionality and that result in the extended instruction sequences \( IS^{left}_{ext} \) and \( IS^{right}_{ext} \). Added operations \( O \) ensure that the pET of the sequence \( IS^{total}_{ext} \) consisting of any of the extended instruction sequences and any subsequent sequence \( IS^{after} \) is equal or higher than the pET of the sequence \( IS^{total} \) consisting of any of the original instruction sequences and \( IS^{after} \).

#### A. Definitions

PUB relies on some of the properties brought by time-randomised caches and the way PTA techniques represent the cache state at any given point in the execution of a program. In this section we present those properties.

PTA techniques keep a probabilistic state of the cache unlike static timing analysis techniques that keep a state of the cache to determine, at any point in the program, which addresses “must”, “may” and “won’t” be in the cache. We call this state of the cache kept by STA techniques [1], deterministic cache state.

During the rest of the paper we assume that the size of the memory unit accessed with a given address \( \Theta_k \), usually known as word, matches the cache line size. The generalisation to the (actual) case in which a cache line may contain several words is straightforward.

**Definition 1:** Probabilistic Cache State (PCS). A given PCS provides the actual probability that any given address \( \Theta_k \) is present in a time-randomised cache at a given instant \( t_i \) of the execution of a program, and thus provides the hit/miss probability for any given address at any point of the execution.

\(*This option requires controlling the compiler backend passes when the executable code is actually generated.*
Definition 2: Probabilistic Execution Time of a Instruction Sequence. We define the probabilistic execution latency (time) of a sequence of instructions, $IS_i$, as the execution time distribution it takes the sequence of instructions to be executed: $pET(PCS_{in}, IS_i, PCS_{out})$, where $PCS_{in}$ and $PCS_{out}$ stand for the PCS right before $IS_i$ is executed and the PCS after its execution respectively. Eventually, we also use $pET(PCS_{in}, IS_i)$ if the PCS after the execution of $IS_i$ is irrelevant (e.g., end of the execution of the program), and $pET(IS_i)$ if, additionally, the initial PCS is irrelevant (or provided by the context).

We say that $pET(IS_i) \geq pET(IS_j)$ if for any cutoff probability the execution time of $IS_i$ is higher or equal than the execution time of $IS_j$. Figure 3 shows the pET of two different sequences $IS_i$ and $IS_j$ where $pET(IS_i) \geq pET(IS_j)$.

Definition 3: Probabilistic Survivability. We define the probabilistic survivability, or simply survivability, of a given access to an address $@k$, at any point in the execution of the program as the probability of hit of that access. It is represented as $Surv(@k)$ and can be approximated by Equation 2.

A cache access may change the PCS of the program that was prior to the execution of such access. This would affect the survivability of all addresses. Given a sequence of cache accesses, the addition of an access $A_1$ to an address $@A_1$, 1) increases or leaves unchanged the survivability of all future accesses $A_2$ to $@A_1$; and 2) decreases or leaves unchanged the survivability of all other addresses. If $@A$ is in cache when $A_1$ executes, it does not alter cache contents since with EoM evictions occur only on the event of a miss. If $@A$ is not in cache, $A_1$ will fetch it, thus increasing or leaving unchanged the probability of hit of any $A_2$, though the eviction caused by $A_1$ can evict other addresses from cache.

B. Instruction Sequence Subordinance

Definition 4: Instruction Sequence Subordinance. Starting from an initial PCS, $PCS_{in}$, a given instruction sequence $IS_i$ is a subordinate instruction Sequence (SIS) of another instruction sequence $IS_j$ if the pET of $IS_i$ is larger or equal than the probabilistic execution time of $IS_j$. That is: $(PCS_{in}, IS_i) \leq_{SIS} (PCS_{in}, IS_j) \iff pET(PCS_{in}, IS_i, PCS_{out}) \geq pET(PCS_{in}, IS_j, PCS_{out})$.

$PCS_{out}$ and $PCS_{out}$ are the PCS left after the execution of $IS_i$ and $IS_j$, respectively.

Definition 5: Instruction Eviction Upper-Bounding. An instruction $A_j$ is an instruction eviction upper-bounding (IEUB) of $B_j$ if the miss probability of $A_j$ is higher or equal than the miss probability of $B_j$. This is the case, for instance, if $A_j$ is known to be a miss (e.g., it has not been accessed before). Next we deal with the general case where $A$ may have been accessed in the past. Let us consider the following sequences: $IS_A = <A_1, ..., A_j>$ and $IS_B = <B_1, ..., B_j>$ where $A_i$ and $A_j$ access address $A$ and no access in between, $\{X_i\}$, accesses $A$; and where $B_i$ and $B_j$ access address $B$ and no access in between, $\{Y_j\}$, accesses $B$. No relation is established between the addresses in both sequences, so addresses may repeat (e.g., $A$ and $B$ could be the same address).

$A_j$ is an IEUB of $B_j$, if we can match all accesses in between $B_i$ and $B_j$ with accesses in between $A_i$ and $A_j$ so that in each pair, the access in the sequence $IS_A$ is exposed to a larger or equal number of evictions than that of the access in sequence $IS_B$. In this scenario, the miss probability of the access of the pair in $IS_A$ is equal or higher than that in $IS_B$. Therefore, the miss probability – and so the pET – of $A_j$ is higher or equal than for $B_j$.

Note that in a EoM cache, the element determining whether a given access misses in cache or not, is the number of evictions carried out in between that access and the previous access to the same address. Further, note that number of evictions that each access suffers depends on the particular initial PCS for each sequence.

An example let us assume the sequences of accesses $<A_1, B_1, A_2, C_1, B_2>$ and $<D_1, E_1, F_1, D_2, E_2>$ and an empty initial cache state, with $X_i$ accessing address $X$. In this case $D_2$ is a IEUB of $A_2$. This is so because we can pair up $B_1$ with, for instance, $E_1$ (both are misses). Similarly, $E_2$ is a IEUB of $B_2$ since we can pair up $C_1$ with $F_1$ (both are misses) and $A_2$ with $D_2$ since $D_2$ has to survive to 2 evictions whereas $A_2$ has to survive to 1 eviction.

Definition 6: Sequence Eviction Upper-Bounding. $IS_i$ is a sequence eviction upper-bounding (SEUB) of sequence $IS_j$ if we can match all accesses in $IS_j$ with accesses in $IS_i$ so that in each pair the access in $IS_i$ is a IEUB of the access in $IS_j$. Then, the pET of $IS_i$ is higher or equal than the pET of $IS_j$. This makes $IS_i$ a SIS of $IS_j$.

SEUB definition is particularly useful to compare sequences whose initial PCS is the same as shown later. The definition of SEUB is related to a similar observation done in the context of probabilistic time composability in [22].

Theorem 1: Starting from an initial cache state $PCS_{in}$, the execution time of the instruction sequence $IS_{orig}$ increases if one access, $C_1$, to any address $C$ – regardless of whether there are other accesses to $C$ in $IS_{orig}$ – is introduced at any point of this sequence, thus obtaining $IS_{ext} = IS_{orig} \cup C$. That is, $(PCS_{in}, IS_{ext}) \leq_{SIS} (PCS_{in}, IS_{orig})$. This is so because $IS_{ext}$ is a SEUB of $IS_{orig}$.

The proof to this Theorem is provided in Section VIII.

Applicability to PUB: The fact that adding cache accesses increases the pET of a sequence allows us to add accesses in the different branches of the CFC until all branches in the extended code include all sequences of accesses in all branches in the original code. Therefore, any branch in the new code will be a SIS of all branches in the original code. This property helps understanding the PUB variant called PUBam (Section IV-A).

C. Cache Subordinance

Definition 7: Unique Address. A given cache access is called unique or said to access a unique address $@un$, if it can be ensured that its survivability is zero. This happens when $@un$ is accessed for the first time or it can be ensured that it was evicted since last time it was accessed, because an invalidation instruction is executed on that address after its last access. For the rest of the discussion, it is assumed that
unique addresses are evicted right after they is accessed, so its
survivability is always zero.

Definition 8: Probabilistic Address Subordinance. A given
PCS$_i$ subordinates another PCS$_j$ for a given access to
address @k if the probability of hit of that access, i.e., its
survivability, is lower or equal in PCS$_i$ than in PCS$_j$. That is,
PCS$_i$(@(k) ⊆ PCS$_j$(@(k) ↔ Surv(PCS$_i$, @k) ≤ Surv(PCS$_j$, @k)).

Definition 9: Probabilistic Cache Subordinance. A given
PCS$_i$ subordinates another PCS$_j$ if for any address, @k, the
survivability of an access to @k is lower or equal in PCS$_i$
than in PCS$_j$, which is represented as follows:

PCS$_i$ ⊆ PCS$_j$ ↔ ∀@k : PCS$_i$(@(k) ⊆ PCS$_j$(@(k)).

If PCS$_i$ ⊆ PCS$_j$, and assuming that core operations
have a fixed impact on pET, then the pET for any sequence
IS$_i = \{I_1, I_2, ..., I_n\}$ is higher when the initial cache state
for IS$_i$ is PCS$_i$ than when it is PCS$_j$. This means that for
any exceedance probability, the execution time is higher or
equal when the initial cache state is PCS$_i$ than when it is PCS$_j$. This is so because under PCS$_i$ every single access
has a lower hit probability than under PCS$_j$, which translates
into an equal or higher pET when starting with PCS$_i$.

Let us consider the sequence of accesses to memory
IS$_{org} = \langle I_1, I_2, ..., I_n \rangle$. In this sequence there is no
constraint on whether each $i (i \in [1..n])$ accesses a different
cache line or one already accessed by another $j$, or whether
data were in cache before.

Theorem 2: Let us assume a given PCS$_{org}$ and the
instruction sequence, IS$_{org}$, whose execution moves the
cache state to PCS$_{org-out}$. Further assume that we replace
an access I by an access, C, to a unique address C
in any position in IS$_{org}$, so that we obtain IS$_{ext} = \langle
I_1, I_2, ..., I_{i-1}, C_1, I_{i+1}, ..., I_n \rangle$. In this scenario: (1) The
pET of IS$_{ext}$ is higher or equal to that of IS$_{org}$, that is, pET(IS$_{ext}$) ≥ pET(IS$_{org}$), and (2) the pET after executing
IS$_{ext}$, PCS$_{ext-out}$, is a subordinate of PCS$_{org-out}$, i.e.,
PCS$_{ext-out}$ ≤ PCS$_{org-out}$ for all addresses.

This theorem talks about the fact that if we replace any
access in a sequence by an access to a unique address (especial
address that cannot be in cache before it is accessed and it is
evicted immediately after being fetched so that further accesses
to C cannot hit in cache), the pET of the IS$_{ext}$ and any
subsequent sequence IS$_i$ is higher than the pET of IS$_{org}$ and
IS$_i$. That is, pET(IS$_{ext} + IS_i$) ≥ pET(IS$_{org} + IS_i$). This
property helps understanding the PUB variant called PUBaa
(Section IV-B).

D. Theorem 1 in Time-Deterministic Caches

In time-deterministic caches Theorem 1 does not apply.
For instance let us assume a fully-associative two-entry cache
deploying LRU replacement. Further assume the sequence
IS$_{org} = \langle A_1, B_1, C_1, A_2 \rangle$, where accesses A$_1$ access
address A, accesses B$_1$ access address B and accesses C$_1$
access address C. Further A, B and C correspond to different
addresses (mapped to different cache lines). That sequence
experiences exactly 4 misses when accessing cache. If we
add one access to address A after B$_1$, so IS$_{ext} = \langle
A_1, B_1, A_2, A_3, C_1, A_2 \rangle$, the new sequence experiences only 3
misses (A$_1$, B$_1$, and C$_1$) despite the insertion of an extra access
since A$_3$ and A$_2$ would hit in cache. As a result, the execution
time of IS$_{ext}$ is shorter than the execution time of IS$_{org}$, while
the cache state left is the same (A and C, with C being the
LRU element).

IV. APPLYING PUB

Based on Theorems 1 and 2, next we show how PUB
extends programs so that they upper-bound the probabilistic
timing behaviour of their original counterparts.

CFC increase the complexity of the WCET estimation using
MBPTA. In order for PUB to produce an upper-bound of
the pWCET of CFCs, requirement PUBReq1 or requirement
PUBReq2 should hold:

PUBReq1. PUB has to derive a bound, pETB, to the
longest pET of the execution of any of the i branches, IS$_{bb}$,
in the CFC plus any instruction sequence IS$_{after}$ after the
CFC. Note that the pET of IS$_{after}$ is affected by the particular
branch that is executed in the CFC. Overall, PUBReq1 imposes
that pETB ≥ pET(IS$_{bb}$) + pET(IS$_{after}$), ∀bb.

PUBReq2. PUB has to upper-bound the longest pET of all
the branches in the CFC and the worst probabilistic cache state
left by any of the branches of the CFC. If this is achieved for
every CFC, the probabilistic execution time of any path in the
extended code is higher than the execution time of any path
in the original code.

To reach its goal PUB deploys different solutions for the
core latency and for cache (either data or instruction) latency.
We focus on cache latency next and we address core latency in
Section IV-D. We start with the application of PUB to the data
cache. Its extension to the instruction cache is straightforward
and it is covered in Annex I.

The following example illustrates the principle of PUB
for the data cache. Example 1: Let us assume an if-then-else
construct prior to which the cache is in a given PCS, PCS$_{in}$.
The purpose of PUB is to extend both branches of the
conditional statement, IS$_{org}$ and IS$_{org}$', into IS$_{ext}$ and
IS$_{ext}'$, such that the impact in the pET of the whole program
of both branches in the extended code, IS$_{ext}$ and IS$_{ext}'$, upper-bounds the pET of the program for both branches
in the original code, IS$_{org}$ and IS$_{org}'$. A.

We propose two different alternatives Address Merging
(PUBam) and Address Aging (PUBaa) respectively.

A. Address Merging (PUBam)

PUBam makes the different branches of the extended
CFC perform at least the same data accesses (same addresses)
and in the same order as any of the branches of
such CFC in the original code. For instance, let us
assume an if-then-else construct as shown in Figure 4(a)
where addresses @A, @B and @C are accessed in the “if”
branch and @D and @E in the “else” branch. In this case,
PUB has to ensure that accesses @A, @B, @C, @D and
@E occur in both branches, making sure that the relative
order of @A, @B, @C, @D and @E is maintained, and so it is
also for @D and @E. Thus, for instance, we could access
@A, @B, @C, @D, @E, or @A, @B, @C, @D, @E, or
@A, @B, @C, @D, @E, but not @A, @B, @C, @D, @E.

While the solution based on replicating all accesses in
all branches of the CFC can always be used, it can be
optimised by avoiding unnecessary replication of accesses.
For that purpose, we identify the longest sequence of accesses
that occurs to the same addresses and in the same order
(although not necessarily consecutively) in all branches, and
do not repeat them. Then, those accesses in the other branch
excluding the repeated ones must be interleaved with the
ones in the current branch in a way that the relative order
of the accesses in the other branches is maintained. This
is better illustrated with an example. Figure 4(b) shows an
example of an if-then-else construct with a sequence


of accesses that repeats in both branches, and how the code could be modified to upper-bound both branches regardless of the branch taken. As shown in the figure, a sequence with 3 accesses in the same order occur in both branches of the if-then-else construct: \(<@A, @C, @G>\). Branches are upper-bounded by properly placing accesses @F, @C and @H in the “if” branch and @B, @D, @A and @E in the “else” branch. Different ways to interleave branches are also valid. For instance, \(<@D, @A>\) and \(<@F, @C>\) could be swapped. In any case, the new sequence of accesses upper-bounds the execution time of the original code in any of both branches in terms of cache behaviour.

Link to Theorem 1: Let us denote the initial cache state before the CFC as \(PCS_{in}\), the sequence of (memory) instructions in any of the branches as \(IS_i\) and the one after the CFC as \(IS_{after}\). We observe that:

- Adding to \(IS_i\) a data access present in any of the other branches of the CFC, leads to the extended sequence \(IS_j\). According to Theorem 1, \((PCS_{in}, IS_i \cup IS_{after}) \subseteq SIS (PCS_{in}, IS_j \cup IS_{after})\), so the total pET of \(IS_j \cup IS_{after}\) increases when \(IS_j\) executes instead of \(IS_i\) since \(IS_j \cup IS_{after}\) is a SIS (subordinate instruction sequence) of \(IS_i \cup IS_{after}\).

- We can repeat this process incrementally adding those accesses in the other branches of the CFC, one by one, keeping the same relative order of those accesses in their original branches. Every time we add a new data access, the resulting sequence is a SIS of all previous sequences, and so of the original branch \(IS_i\).

- Given that for each branch, \(b\), we add all the accesses that are in the other branches but not in \(b\), also makes the extended version of each branch be a SIS of all original branches.

This process can be carried out using a bottom-up approach starting from the innermost CFC until the whole program is analysed adding the corresponding data accesses. By doing so \(PUBReg\) is met.

Different CFC. So far we have considered simple if-then-else constructs. Next, we consider other types of CFC such as if-then, switch, loop and nested if-then-else constructs.

If-then. An if-then construct can be treated analogously to if-then-else ones assuming that the “else” branch is empty, so that the accesses in the “if” branch are simply added in the new “else” branch.

Switch. In switch (or if-then-elsif-elsif-...) constructs, the number of branches can be larger than 2. The sequence of repeated access should, therefore, exist in all branches. All accesses in all the remaining branches excluding the repeated sequence (repeated in all branches) are placed appropriately by PUB in the current branch.

Nested Conditionals. Regarding nested CFC, our mechanism can be applied recursively starting from the innermost CFC. When moving one level up, since all the branches of the innermost CFC are trustworthy upper-bounds of the original branches, any of them can be used. When a number of instructions is to be inserted in one branch of the outer CFC, they must be inserted identically in all branches of all inner CFC. An example of nested CFC, with if-then-else and if-then constructs is shown in Figure 4(c). The first column shows the original code. The second column the branch upper-bounding for the innermost if-then that becomes if-then-else. Finally, the last column shows the code after branch upper-bounding for the outermost CFC. As shown, the “if” branch of the outermost if-then-else construct must be also extended with the accesses in the if-then-else construct inside the “else” branch.

Loops. Branch upper-bounding is trustworthy even in the case of loops because although data cache accesses may change across iterations (e.g., vector traversals), accesses in any of the branches of the CFC remain the same. Note that not replicated accesses are only those that can be guaranteed to access the same address in all branches of the CFC in every iteration. For instance, if two accesses occur to a position in an array in the different branches of a CFC and they are guaranteed to access the same position (e.g., \(a[i]\) where \(i\) is the loop index), then the mechanism is trustworthy. If those accesses cannot be proven to access the same position in all iterations (e.g., \(a[i]\) and \(a[j]\)), then they are assumed not to access the same address and are replicated in the different branches of the CFC.

Infeasible paths, error codes and modes of operation. Unless instructed otherwise \(PUBam\) balances the different branches on every CFC. However, there are several circumstances in which one or several of the branches of a CFC do not need to be considered in the computation of the pWCET, or at least do not have to be considered together. For example, if the user deems some
CFC as non-relevant for the WCET because their execution does not affect the WCET (e.g., code to deal with error conditions), the user can instruct PUB, e.g., by means of annotations, not to balance instructions in those branches, thus reducing PUB overhead.

Operation modes is another example in which the user can help PUB. Software with different operation modes, which can be mutually exclusive, encapsulates different functionalities for each of which a different pWCET can be derived to reduce the pessimism that an across-modes pWCET would incur. In order to prevent PUB from generating a pWCET that upper-bounds all those together, the user must correctly identify them and indicate to PUB the code belonging to different operation modes. Then the user can provide path coverage for those operation modes (i.e. one input per mode). For instance, in a switch statement in which each case represents a different mode, it would be simple to annotate the code to prevent PUB from balancing the different branches of the switch.

Function Calls. If a function is called in one branch of a CFC, its effect on the cache has to be replicated on the other branches. This can be done either (1) by creating a dummy function that accesses the same addresses in the same order, or (2) by means of the Address Aging technique presented next. Further, if the function is called in all branches of a CFC with the same inputs then the function can be considered the ‘common pattern’ in all branches of the CFC, applying PUB only to the code before the call and after the call to the function.

B. Address Aging (PUBaa)

The fact that PUBam requires merging accesses of all branches of a CFC may lead to pessimistic pWCET estimates, which we call PUBam inefficiency. This pessimism manifests itself when the code in any branch of a CFC in the extended version has significantly higher number of accesses than that in any branch of the original version. This results in a PCS after the execution of the extended version of any of the branches of a CFC that is worse (i.e. the hit probabilities of any subsequent accesses is lower) than the worst PCS obtained after executing any other branch of the original CFC.

PUBam introduces low overhead on if-then-else CFCs in which in one of the branches \( b_{1-\text{org}} \) few accesses to cache are carried out. In that scenario the addresses of the other branch \( b_{2-\text{org}} \) can be copied into \( b_{1-\text{ext}} \) leading to \( b_{1-\text{ext}} \) with \( b_{2-\text{ext}} \) almost unchanged with respect to \( b_{2-\text{org}} \) since \( b_{1-\text{org}} \) contains few accesses. In this case the worst cache behavior in the extended version, which happens when \( b_{2-\text{ext}} \) executes, is almost the same as when \( b_{2-\text{org}} \) executes. Similarly, PUBam introduces little overhead if both CFC branches perform similar access sequences as few extra accesses need to be added. Note that PUBam also handles efficiently if-then-else constructs, which represents the case when one of the branches is empty as PUBam keeps the worst path untouched and only increases the code in the missing path.

PUBam is inefficient when in each branch of a CFC different sets of addresses are accessed. In that case, all branches have to be extended adding the accesses present in the others, making that the execution time and the PCS after executing any of the extended branches are much worse than those obtained after executing any of the original ones. For instance, if we have a 10-case switch such that in each of its branches 2 different addresses are accessed, this would result in an extended switch containing 20 accesses in each branch.

In order to handle this inefficiency of PUBam we propose a new technique called address aging (or PUBaa). Instead of copying addresses of one branch on the rest, PUBaa adds accesses to addresses accessed nowhere else in the program, which lead to a miss and fetch no useful data. In particular, it adds as many accesses as the maximum number of accesses in any of the branches. In the previous example of a 10-case switch in which each branch has 2 different accesses, in each branch PUBaa adds 2 accesses to unique addresses.

Link to Theorems 2 and 1: The idea is that in the original switch the worst situation happens when the branch executed incurs two misses. If we consider only the 2 added accesses which miss and fetch no useful data, then their execution time and the PCS they leave are worse than those in the original code in all branches. If we consider that those two accesses virtually replace the accesses in the other branches of the original program, Theorem 2 applies, meeting PUBReq2. The fact that in the branch the two missing accesses are added back (those in the original code) only increases the execution time according to Theorem 1, hence meeting PUBReq1.

Note that PUBaa is particularly good when the number of branches in the CFC is high. Conversely, if the number of branches is low and they potentially present a high imbalance, PUBam is more efficient. This makes PUBam and PUBaa complement each other.

PUBaa can be implemented with or without hardware support. If no hardware support is in place, accesses added by PUBaa are ensured to miss by accessing addresses never accessed before. A data structure (\( \text{dummy} \)) and a pointer (\( \text{next} \)) are created so that the address accessed by any of those accesses is the one pointed by \( \text{next} \) (so \( \text{dummy}[\text{next}] \)) and \( \text{next} \) is increased by the cache line size of the affected caches. Alternatively, PUBaa can be implemented with hardware support by adding a special instruction, MissPUBaa that creates a miss in cache, hence creating a random eviction, and accesses memory, but that brings no useful content to cache.

C. Creating the PUB Code

Adding memory operations can be done in several ways depending on the hardware characteristics. If a non-modifiable register exists (e.g., register \( r0 \) in SPARC and MIPS architectures, which is hardwired to zero), new accesses can be translated into \( r0 = \text{LOAD} \ A \text{_{A}} \), where \( A \text{_{A}} \) is the particular address to be accessed. If such a register does not exist, then a free register must be used to hold the data read from memory. Such data will not be used, but the semantics of the instruction set architecture (ISA) might not support reading a pointer or reading data from memory. Finally, note that in some cases accessing an address \( A \text{_{A}} \) that was not accessed in the original program may create an exception. This is not often the case since most memory operations can be guaranteed not to access beyond the memory bounds of the program. For the remaining accesses, if PUBaa is applied, as mentioned before, it is required creating a data structure so that unique accesses to that structure age the cache state appropriately if no hardware support is in place.

Code Alignment. When applying PUB, code (instruction accesses) may get unaligned with respect to the original code, thus altering the timing behaviour in a way that may end up not upper-bounding the original code. It is important realising that the survivability of a given access, i.e. its hit probability, does not depend on the particular address it is mapped in memory. The address in a deterministic cache would determine the particular set in which the address is mapped in cache, but this is not the case in a time-randomised cache, since the placement is randomised breaking the dependence between the address of an access and its assigned cache set. Given two accesses to cache what really affects their hit/miss probability is whether
they are mapped to the same cache line or not, which in turn affects their reuse distance (represented as $k$ in Equation 2). Hence, a principle we follow when applying PUB is not to unalign basic block boundaries, so that, instructions that in the original code are mapped to the same line out of the basic block being modified remain mapped to the same line in the extended code and vice-versa. Also, the code in the basic block – whose all instructions are always traversed sequentially – keeps the same cache-line alignment at its boundaries so that it has the same behaviour as is the original code increased with the access to some extra cache lines. This is achieved by ensuring that PUB-added instructions are added so that the total size of the code added in each of the branches of the CFC is a multiple of the cache line size. This makes that the rest of the code keeps exactly the same cache-line alignment as in the original code. Again, note that altering the actual addresses of the pieces of code is irrelevant given that PTA-compliant designs are built upon random placement caches [8] or software randomisation is used on top of deterministic placement (e.g., modulo placement) caches [23].

D. Core latency

We focus on a processor architecture deploying time-randomised instruction and data caches with a core pipeline similar to the LEON4 [16] processor in which processor instructions have a fixed latency, as it has been shown to be PTA-compliant [8]. Core operations, besides their access to the instruction cache, which is covered in Annex I, do not affect the PCS of the data cache. PUB computes the core latency of the instructions in the different branches of a CFC and adds core instructions to each branch such that the core latency of all branches is equalised. PUB adds the maximum number of instructions taking into account the timing effect that they introduce. Added instructions will be typically arithmetic-logic operations, such as integer or floating point additions, multiplications, etc. that have a neutral effect on the functional behaviour of the program by either writing the same value read (e.g., multiplying by 1) or writing into a hardwired constant-value register.

E. Steps

PUB carries out several steps that should be applied in the following order: First, core latency and data cache latency are upper-bounded in any order. Then, instruction cache latency is upper-bounded (excluding code alignment) for efficiency reasons, since it can make use of those instructions already added by the previous steps. Finally, code alignment is performed.

V. EVALUATION

We model a 4-stage pipelined core architecture in which instructions are fetched in-order from the Instruction cache (I-cache) into the processor. Instructions are decoded and then issued to the execution stage, where they are executed in a fixed latency, except for memory operations. Load and store operations access the data cache (D-cache) during this cycle. Our architecture ensures that requests sent to each resource are issued in program order and are also served in program order to prevent timing anomalies by construction [24].

The size of both the I-cache and the D-cache is 8-KB with 16-byte lines and 8-way set-associativity. Both caches, instruction and data, implement random placement and replacement policies [8]. The D-cache is write back. On a dirty line eviction the pipeline is stalled. In our architecture stores may have lower impact on execution time than loads due to the use of a store buffer. To handle this PUB added memory operations are loads.

The latency of the fetch stage depends on whether the access hits in I-cache (1 cycle) or misses (100 cycles). After the decode stage, memory operations access the D-cache whose behaviour is analogous to that of I-cache.

We use benchmarks from the EEMBC Autobench [25] and Mälardalen [26] suites as reference for the analysis. EEMBC Autobench is a well-known benchmark suite that reflects the current real-world demand of some embedded systems. Mälardalen benchmarks [26] are also commonly used in the community to evaluate and compare different types of WCET analysis tools and methods.

For several of these benchmarks we have the input set leading to the worst-case path. For some of them, the number of iterations they carry out depends on the input data. The input data we have for them provide the worst-case path for the a given loop-iteration count that we assume fixed in this paper. The benchmarks in this group are: bs, cnt, fir, ldcnum, prime, insersort, edn, matmult, janne, cover, f dct, fibcall, jfdctint. For these benchmarks, the ratio between the WCET estimation with PUB and the one with MBPTA corresponds to the actual WCET overestimation introduced by PUB. For the rest of the benchmarks, for which we could not derive the worst-case path, PUB WCET overestimation is an upper-bound of its actual WCET overestimation, since another input set could exist that exercises the worst path, making MBPTA pWCET estimate to increase and be closer to the one provided by PUB.

A. Code Replication Size

It is important to note that with PUB the extended program can be used only at analysis time to derive pWCET estimates, while at deploy time the original program can be used.

Figure 5 shows the code-size overhead introduced by PUB. We observe that for most of the benchmarks the overhead is below 20%, with only few benchmarks introducing higher overhead. The case of few EEMBC benchmarks (e.g., puwmod, matrix and a2time) is remarkable since the code size is increased noticeably. In contrast, some Mälardalen did not require any change (e.g., cover, edn, f dct, fibcall, insersort, janne, jfdctint, matmult and ns) since they do not have any conditional CFC or those conditional CFCs exist but they are already fully balanced (e.g., cover has some constructs where a given variable is incremented or decremented in different branches but cache is not accessed).

Regarding those benchmarks experiencing a significant code size increase due to PUB, we have observed that this highly correlates with the number of conditional CFCs and their degree of nesting. For instance, the four EEMBC benchmarks with lowest number of conditional CFCs (up to 6) are aiffir, aiffir, aiffif and cacheb, and are the ones experiencing the lowest code increase. Conversely, puwmod is the benchmark with highest number of conditional CFCs (66) and frequent conditional CFC nesting (often 3 conditional CFCs nested).

B. pWCET estimates

In this section we compare the pWCET estimates obtained with PUB and MBPTA. Note that a first comparison of PTA techniques and static-timing analysis techniques can be found in [27]. Following the iterative method in [3] we carried out 1,000 experiments, which proved to be enough according to MBPTA [3]. MBPTA then uses those execution time measurement and EVT to extract pWCET estimates. In all experiments
we apply the Wald-Wolfowitz independence test and the two-sample Kolmogorov-Smirnov identification distribution test. We use a 5% significance level (a typical value for this type of tests). We also apply the ET test for Gumbel convergence testing. All three tests were passed for all EEMBC and Mälardalen benchmarks.

PUBaa exploits the case in which there is little overlap between the (long) address sets in each branch of a CFC. This would lead to PUBam generating long extended versions of each branch with many accesses to all those addresses, which in fact generates worse PCSs than feasible with the original version of the code. Such a case has not been found in the benchmarks evaluated in this paper, making PUBaa not to have any benefit over PUBam. As part of our future work, we aim at finding benchmarks that allow us to illustrate quantitatively scenarios where PUBaa outperforms PUBam.

Figure 6 shows the pWCET increase introduced by PUB with respect to the original MBPTA. We observe that for EEMBC and Mälardalen the average pWCET slowdown is 11% and 5% respectively. Only three benchmarks suffer a slowdown between 20% and 50%. a2time, for instance, uses a data working set fitting quite well into the D-cache, so when applying PUB more data are accessed on each iteration, the working set exceeds cache size and execution time (and so pWCET) increases noticeably. However, most of the benchmarks experience low or even negligible pWCET degradation when applying PUB.

Similar to other timing analysis techniques, PUB benefits from information on infeasible paths. In particular, for a2time, putptrch and ns, we have used semantic knowledge of the infeasibility of some paths. We detected that some path combinations were infeasible by semantic construction of the program. This is the case of a2time, detailed in Annex II, in which there is a sequence of if-then constructs out of which exactly one can be executed in any traversal. Therefore, since every execution is indeed triggering the worst (and only) timing behaviour, there is no need for balancing those if-then constructs. If this semantic information is not provided by the user, PUB assumes that all CFC can be executed in each traversal, thus producing some non-negligible and ‘artificial’ overhead. If the user prevents PUB from balancing CFC in which some path combinations are deemed to be infeasible the overheads of PUB can be reduced.

VI. RELATED WORK

Several approaches have been proposed to deal with the complexity of WCET analysis when different input data cause the code to execute on different execution paths with differing execution times. The Single-Path approach [28][29] transforms CFC into a sequential set of instructions based on the concept of predic ted execution [30]. Predicated instructions have a predicate, such that the instruction is only executed if its predicate evaluates to true, otherwise the processor assumes it to be a nop operation. For instance, for an if-then-else, the instructions in each branch are changed by sequential code with conditional-move assignments for each of the conditionally changed variables. PUB does not make the execution of a program single-path, but instead changes the branches in conditional CFCs such that extended paths take longer to execute than any of the original branches of the conditional CFC. Further note that the extended code can be only used at analysis time; at deployment time the original code can be used, hence adding no overhead on average performance.

Several studies such as [31] focus on probabilistic schedulability analysis and assume, for each task to be scheduled, a known distribution of execution times. Other studies focus on WCET estimation such that WCET estimates are probabilistic, which is the focus of this paper. Some of those studies focus on time-deterministic architectures and assume that the external events affecting the execution time of a program, i.e. input data vectors, are assumed to be random variables. Based on this assumption authors associate to each potential execution path of the program a probability. To this end, each condition is associated a probability. For instance in the statement if (a > b) then S1; else S2; if a and b are input data, these approaches assume that it is known the probability of the condition to be true, and hence the probability S1 to be executed, can be computed. This puts high requirements on the user to provide input data that is probabilistically representative of deployment time behaviour. In that respect, it is worth mentioning that the primary goal of MBPTA is to provide pWCET estimates that hold under execution conditions that may occur during actual operation: whereas those conditions may not be exactly identical to those captured by the observation runs made at analysis time, analysis time conditions must still reproduce or upper-bound the probabilities of the execution times that may occur during operation [10]. MBPTA controls the hardware and the software behaviour, for instance by proposing time-randomised caches, so that the requirements on the user to provide representative execution conditions is heavily reduced [10].

Other authors [32] let the application run long enough on a time-deterministic architecture collecting the observed execution times. Those execution times are then randomly sampled to apply Extreme Value Theory. However, the representativity of the obtained results (i.e. pWCET estimates) depends on how representative the collected observed execution times are, with respect to the application’s execution times, and those techniques provide no guarantees on that matter [10].

Authors in [33] define the concept of probabilistic cache behaviour on a deterministic fully-associative cache. To that end, for every point in the program the (static) deterministic cache state in which the program is at that point is associated the probability of reaching that program point. On the one hand, authors also assume that the events affecting the execu-
tion time are random so that a probability can be associated to each path with the casuistic shown above. On the other hand, this differs from our concept of probabilistic cache state, in which each access has a distinct associated probability of hit and execution paths have no associated probability.

VII. CONCLUSIONS

Obtaining trustworthy and tight worst-case execution time (WCET) estimates is of prominent importance in safety-critical real-time embedded systems. However, the WCET estimation process has to be affordable for the user to apply it. Measurement-Based Probabilistic Timing Analysis (MBPTA) has emerged recently to respond to this challenge by providing means to easily determine accurate WCET estimates. Unfortunately, MBPTA still relies on the user providing path coverage, which is often beyond of what the user can provide.

In this paper we propose a Path Upper-Bounding method, PUB, which works in conjunction with MBPTA. PUB allows deriving pWCET estimates for all paths of the program, including those not exercised by the input vectors given. This makes that the coverage needed for functional verification is also enough for the timing analysis, thus drastically reducing the cost of timing analysis. Our results show that PUB increases pWCET estimates only by 5% and 11% on average for Mälardalen and EEMBC with respect to MBPTA.

ACKNOWLEDGMENTS

The research leading to these results has received funding from the European Community’s Seventh Framework Programme [FP7/2007-2013] under the PROXIMA Project (www.proxima-project.eu), grant agreement no 611085. This work has been also supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557, the HiPEAC Network of Excellence, and COST Action IC1202: Timing and i.i.d property. White Paper.”

REFERENCES

any sequence of accesses to addresses other than A and C, may observe some gains, i.e., reduction in pET, for A2 when
C1 is added before C2. Note that we focus on the case where
A2 misses in ISorig and potentially hit in ISext (if A2 were a
hit in ISorig the inclusion of C1 would not reduce its latency
any further). Despite that, it is still the case that, ISext is a
SEUB of ISorig. In both sequences we ignore the effect on
the other accesses other than A1 and A2. Once the whole
analysis is complete, it can be extended to any other access.
We identify the following scenarios:
• If C1, which misses in cache, is introduced between the
two accesses to A, leading to the instruction sequence
ISext = < A1, ..., C1, ..., C2, ..., A2 > and this may
make C2 become a hit, still A2 in ISext will suffer as
many evictions in ISext as in ISorig – at least one in the
former and exactly one in the latter, see Figure 7a. Hence
A2 in ISext is a SEUB A2 in ISorig. As before C1 in the
extended sequence is a SEUB of C2 in the original.
• If C1 is added before A1, hence leading to ISext = <
C1, ..., A1, ..., C2, ..., A2 >, we identify two cases: (1) A2
misses in ISorig, because it is evicted by an access B1,
with B1 and C2 accessing different addresses; and (2)
and no access B1 evicts A2 and C2 may evict A2.
  – Under (1) whether C2 becomes a hit due to the
inclusion of C1 has no effect on A2. As a result, C1
and A1 in ISext are IEUB of C2 and A1 in ISorig,
while A2 remains being a miss (see Figure 7b).
  – Under (2), we identify two final sub-scenarios: (2.a)
A1 is a miss in ISorig and (2.b) it is a hit.
Under (2.a), we can match C1 and A1 of ISext
with C2 and A1 of ISorig, since all of them are misses.
We can also match C2 of ISext with A2 of ISorig,
as they suffer one eviction — C2 suffers the eviction
causally by A1 in ISext and A2 suffers the eviction
caused by C2 in ISoriginal (Figure 7c).
Under (2.b), we can match C1, A1 and C2 of ISext
with C2, A2 and A1 of ISorig respectively as C1
(ISext) and C2 (ISorig) miss both. A1 (ISext) and
A2 (ISorig) suffer one eviction each (by C1 and C2
respectively), and C2 (ISext) can be a hit in the best
case and A1 (ISorig) is a hit (Figure 7d).

In summary, in all cases adding an access C1 to the
original sequence ISorig makes that the new sequence, ISext,
becomes a SEUB of ISorig, and so have a higher pET. Hence,
ISext, where ISext = ISorig ∪ C1, is a SIS of ISorig as
pET(ISext, ISorig) ≥ pET(ISext, ISorig).

We further illustrate this with an example of a com-
plex sequence. Let us assume ISorig = < A1, B1, C1, D1, B2, C2, A2 > and ISext = < A1, D1, B1, C1, D2, B2, C2, A2 > after adding D1. The initial
PCS is empty. We can match the following pairs from
[ISext, ISorig]: [A1, A1], [B1, B1], [C1, C1], [D1, D2] as all of them are misses; [D2, B2] as they suffer both 2 evictions
(and they generate δ evictions); [B2, C2] as they suffer 1 + δ
evictions (and generate γ evictions); and [A2, A2] as
they suffer 3 + γ evictions in ISorig and 3 + γ + θ
in ISext where θ is the number of evictions produced by C2
in ISext. Overall, ISext is a SEUB of ISorig.

Proof Theorem 2: Obviously, the survivability of the
accesses executed before C1 is not affected by C1. Since
C is not present in cache, access C1 introduces exactly
one eviction in a random line in cache – with each line having
a probability to be evicted. This decreases, or does not affect, the
survivability of subsequent accesses. The assumption that C

\[
\text{Fig. 7. Illustration of possible cases in Proof 1. For each access, e.g. A1, the superindex indicates whether it is a hit A^\text{hit} or a miss A^\text{miss}. Arrows shown which access in each sequence can be paired up according to IEUB definition is evicted after it is accessed makes that no future access to C benefits from it. Further, the execution of C (miss latency) is equal or higher to that of the access it replaces, thus increasing (or leaving unmodified) the pET of the extended sequence, making pET(IS_{ext}) ≥ pET(IS_{orig}).}
\]

ANNEX 1: PUB FOR INSTRUCTION CACHES

The same principles applied to the data cache apply to the
instruction cache, although the instruction cache has several
peculiarities. Three main aspects are considered by PUB to
deal with the instruction cache: code invocation, function calls
and code alignment.

Code Invocation. For the sake of this explanation let
us assume an if-then-else construct where IS_{left} is
executed in the “then” branch and IS_{right} in the “else” branch.
We would like to apply the same solution as for data, where
accesses in other branches of a CFC are reproduced in the current
one. Ideally, this could be conceptually achieved if we could
access all the instruction addresses of the CFC, every time any
branch of the CFC is executed. In the if-then-else, if we
were to create IS_{join} = IS_{left} ∪ IS_{right}, then IS_{join}
would be a SIS for both IS_{left} and IS_{right}, or formally stated:

\[
\begin{align*}
\text{PCS}_{in, IS_{join}} & \subseteq \text{SIS}(\text{PCS}_{in, IS_{left}}) \\
\text{PCS}_{in, IS_{join}} & \subseteq \text{SIS}(\text{PCS}_{in, IS_{right}})
\end{align*}
\]

Then, we would like to have IS_{join} in both branches of
the if-then-else construct so that the whole if-then-else
construct in the extended version CFC_{ideal} <- if IS_{join} else IS_{join} > is a SIS
of the whole if-then-else construct in the original version
CFC_{org} <- if IS_{left} else IS_{right} >.

However, this is not possible because the only way to access
an instruction address is to fetch and execute the instruction
placed in this memory location. In the case of the CFC this
requires to modify the control flow of the execution such
that both branches of the CFC are executed always, which
would have undesirable functional effects. Since the particular
addresses are irrelevant for time randomised caches, we can
produce the same timing effect by building CFC_{real} <- if
IS_{join-\text{left}} else IS_{join-\text{right}} > where IS_{join-\text{left}} = IS_{left} ∪ IS_{right}' and IS_{join-\text{right}} = IS_{right} ∪ IS_{left}', such
that IS_{left}' and IS_{right}' have the same pattern of access to
the instruction cache as IS_{left} and IS_{right} respectively. Later
we explain how to make IS_{left}' and IS_{right}' not to have any
functional effect.

If CFC_{real} is executed just once, its timing behaviour will
be exactly the same as for CFC_{ideal}.

If the CFC_{ideal} is executed several times (which is
the case in a loop), being IS_{1}, IS_{2}, ... the sequences
of instruction executed after every time CFC_{ideal} runs,
then we would have the following sequence of execution:

\[
\text{IS}_{\text{join}}, IS_{1}, IS_{\text{join}}, IS_{3}, IS_{\text{join}}, IS_{5}, IS_{\text{join}},...
\]

\[
\text{PCS}_{in, IS_{join}} \subseteq \text{SIS}(\text{PCS}_{in, IS_{left}}) \\
\text{PCS}_{in, IS_{join}} \subseteq \text{SIS}(\text{PCS}_{in, IS_{right}})
\]

\[
\text{PCS}_{in, IS_{join}} \subseteq \text{SIS}(\text{PCS}_{in, IS_{left}}) \\
\text{PCS}_{in, IS_{join}} \subseteq \text{SIS}(\text{PCS}_{in, IS_{right}})
\]
With $CFC_{real}$, any of the branches $IS_{join-left}$ or $IS_{join-right}$ can be executed instead of $IS_{join}$. Hence, if in the different runs of the CPC, the same branch is always executed, the reuse distances and so the hit/miss probabilities – and so the pET – will be the same as in the ideal case where $IS_{join}$ is executed always. However, if different branches are executed as, for instance, in the following sequence: $IS_{join-left},IS_{join-right},IS_{join-right},IS_{join-right},IS_{join-left},\ldots$ then, reuse distances grow w.r.t. $IS_{join}$. As a consequence, any sequence of paths of $IS_{join-left}$ else $IS_{join-right}$ experiences the same execution patterns as if $IS_{join}$ else $IS_{join}$, but with equal or higher reuse distances, and thus with equal or higher pET. Therefore, if $IS_{join-left}$ else $IS_{join-right}$ is a SIS of if $IS_{join}$ else $IS_{join}$, which is already a SIS of if $IS_{left}$ else $IS_{right}$.

So far we have assumed that $IS_{left}$ and $IS_{right}$ do not have any effect on the functional behaviour of the program. This challenge can also be addressed easily by making sure that the added instructions do not impact the functional behaviour – for example making write operations access behaviour — for example making write operations access.

This challenge can also be addressed easily by making sure that the added instructions do not impact the functional behaviour of the program.

7A nop, no-operation, instruction is any instruction producing no functional effect in the execution of the program. In some ISAs such an instruction exists. If this is not the case, it is always possible performing operations like adding ‘0’ to any particular register.

---

**Annex II: a2time EEMBC Benchmark or How to Exploit User Knowledge on Infeasible Paths**

In this section we discuss the exploitation of user knowledge about the application in order to reduce the PUBam overhead which is caused by infeasible paths in a program.

Applications can have path combinations that are not possible to be exercised under any circumstances at deployment time. An example of such combinations are paths that come from different operation modes in the application, e.g., take-off and cruise mode. Another case appears when there are conditional paths in the program, whose conditions are impossible to be satisfied at the same time. In both cases, when applying timing analysis in an application agnostic way, if the program’s Worst Case path includes paths from infeasible path combinations, it will result in an overestimation of the WCET. To overcome this problem the WCET either needs to be computed in a per-operation mode manner, or by not taking into account those paths that cannot be taken together. As an example of this behaviour we present the a2time benchmark of the EEMBC automotive suite.

The structure of this program follows the pattern of EEMBC automotive benchmarks: a loop with a specified number of iterations. The loop body is unrolled explicitly 3 times, which makes the sequence of input sensing and calculations inside the loop body to be repeated 3 times.

In Figure 8 we can see the pseudocode of the a2time’s loop body. The compiler cannot know that the 8 paths are mutually exclusive, which is actually the case. Therefore PUB will balance all paths with an else construct. Since exactly one of the 8 conditions is true, this will cause the program to execute 7 expensive else paths unnecessarily (multiplied by 3 as the loop is unrolled 3 times), as those paths are infeasible. This results in a 2.3X times increase in the execution time.

Users knowledge about the application can be used to reduce this overhead. This can be done by: a) using compiler directives, in order to instruct the compiler not to balance these paths; or b) restructuring the code to make clear to the compiler that these paths are exclusive to each other.

The former requires using compiler directives (e.g., #pragma no_balance) and has the advantage that keeps the application unmodified, which is of particular benefit in case of legacy applications that are already certified.

The latter consists in a simple restructuring of the application in order to make explicit that the paths are infeasible. For example, in Figure 9, the individual if statements are changed to else if conditions, which shows that only one of the 8 paths can be true at any moment. This way, PUBam would only add a single balance path in this construct, which would be never exercised, so no overhead would be induced. Application of PUBam to the rest of the paths in the benchmark, leads to a 47% increase in the WCET, compared to the 2.3X without taking advantage of knowledge about the application.

---

Fig. 8. a2time loop structure

```plaintext
for i < ITERATIONS do
  if angleA ≤ angle < angleB then
    firing_time = complex math expression 1 // Cylinder 1
  end if
  if angleC ≤ angle < angleD then
    firing_time = complex math expression 2 // Cylinder 2
  end if
  \ldots
  if angleO ≤ angle < angleP then
    firing_time = complex math expression 8 // Cylinder 8
  end if
end for
```

Fig. 9. a2time loop structure with simple code restructuring to help the compiler identify mutual exclusive paths

```plaintext
for i < ITERATIONS do
  if angleA ≤ angle < angleB then
    firing_time = complex math expression 1 // Cylinder 1
  else if angleC ≤ angle < angleD then
    firing_time = complex math expression 2 // Cylinder 2
  else if angleO ≤ angle < angleP then
    firing_time = complex math expression 8 // Cylinder 8
  end if
end for
```

---

**Annex II: a2time EEMBC Benchmark or How to Exploit User Knowledge on Infeasible Paths**