Software Time Reliability in the Presence of Cache Memories

Suzana Milutinovic[1,2], Jaume Abella[1], Irune Agirre[3], Mikel Azkarate-Askasua[3], Enrico Mezzetti[1], Tullio Vardanega[4], and Francisco Cazorla[1,5]

[1] Barcelona Supercomputing Center (BSC), Barcelona, Spain
{suzana.milutinovic,jaume.abella,enrico.mezzetti,francisco.cazorla}@bsc.es
[2] Universitat Politècnica de Catalunya, Barcelona, Spain
[3] IK4-IKERLAN, Arrasate-Mondragòn, Spain
{iagirre,MAzkarateAskasua}@ikerlan.es
[4] University of Padova, Padova, Italy
tullio.vardanega@math.unipd.it
[5] IIIA-CSIC, Barcelona, Spain

Abstract. The use of caches challenges measurement-based timing analysis (MBTA) in critical embedded systems. In the presence of caches, the worst-case timing behavior of a system heavily depends on how code and data are laid out in cache. Guaranteeing that test runs capture, and hence MBTA results are representative of, the worst-case conflictive cache layouts, is generally unaffordable for end users. The probabilistic variant of MBTA, MBPTA, exploits randomized caches and relieves the user from the burden of concocting layouts. In exchange, MBPTA requires the user to control the number of runs so that a solid probabilistic argument can be made about having captured the effect of worst-case cache conflicts during analysis. We present a computationally tractable Time-aware Address Conflict (TAC) mechanism that determines whether the impact of conflictive memory layouts is indeed captured in the MBPTA runs and prompts the user for more runs in case it is not.

Keywords: Probabilistic Timing Analysis · WCET · Representativeness · Cache memories

1 Introduction

Measurement-based timing analysis (MBTA) is widely adopted in the real-time domain [22]. The obtained worst-case execution time (WCET) estimates, however, are reliable insofar as the user is capable of designing test scenarios whose conditions are close to those that can arise during operation. Complex hardware and software, e.g. caches, introduce numerous sources of jitter (*soj*) that are difficult to analyze and control. For example, how program objects, such as code or stack, are assigned to memory defines their memory addresses, which in turn determines how they are mapped to cache sets and, ultimately, the program's pattern of hits and misses. Controlling the effect of memory layout

to avoid incurring bad scenarios is not always feasible in practice. Existing techniques are typically exploitable only at the end of the development process as any analysis result obtained on single software units gets inevitably disrupted after integration. This inherently clashes with the principle of incrementality in software development and analysis, which is a fundamental cross-domain industrial concern [16].

Measurement-Based Probabilistic Timing Analysis (MBPTA) [6, 2, 21] exploits Extreme Value Theory (EVT) [14] and time-randomization to increase the confidence on WCET estimates. MBPTA uses EVT to model the probability of extreme events and, in particular, the combined probability of the events whose impact is captured in the execution time observations. EVT treats the system as a black box, focusing just on its output, hence providing no help to derive an argument of whether all *soj* are properly covered. And here is where time randomization comes to the rescue: higher coverage of *soj* can in fact be obtained by injecting time randomization in the operation of complex jittery resources to replace hard-to-control deterministic behavior, so that the corresponding *soj* exhibit probabilistic behavior. Interestingly, this feature also allows reasoning on whether enough measurement runs have been made, which will be the case when the residual probability of missing a significant behavior of the *soj* becomes provably negligible. For instance, if the extreme behavior of a *soj* has a probability of appearance of $P_{event} = 0.1$ per run, the probability of not observing it in $R = 1,000$ runs is $P_{nobs} = (1 - P_{event})^R = (1 - 0.1)^{1000} = 1.7 \times 10^{-46}$.

Time-randomized caches (*TRc*) [11] are MBPTA's preferred cache designs and have been demonstrated on FPGA implementations [9]. *TRc* use random placement, mapping memory addresses to random cache sets at each run, giving rise to random *cache (set) placements* across runs. As in deterministic set-associative caches, when the number of addresses mapped to a cache set exceeds its associativity (W), systematic cache conflicts may occur and eventually result in increased execution times. With *TRc* we do not need to control the cache mapping to avoid or trigger some specific scenarios, as the effect of cache placement is transparently exposed. Still, it must be guaranteed that the effect of placement is conveniently captured at analysis time. And this is not given since, *conflictive cache placements* may occur with a probability high enough to impact the timing budget of the system, but low enough to defy observation in the analysis runs [3, 20, 17]. For example, for an application that accesses 5 addresses in its execution, the probability that all of them are randomly mapped to the same set in a 32-set 4-way cache is $10^{-6} \approx (1/32)^4$, which can be of relevance for the domain safety standards. If $R = 1,000$ analysis runs are performed, a typical value for MBPTA, the probability of mapping the five addresses in at least one run to the same set is very low ($\approx 10^{-3}$). So far, this issue has been solved in limited scenarios, which assume that either the program addresses memory uniformly [3] or it accesses a small number (≤ 15) of cache lines [18].

In this paper we present the Time-aware Address Conflict (TAC) approach, a general and computationally-tractable method that, from the program's sequence of accessed addresses, determines whether the number of runs performed by

MBPTA, referred to as R, suffices to capture conflictive cache combinations with sufficient probability. Else it derives a higher number of runs, referred to as R', for which this can be asserted. TAC derives a list of address combinations that, when mapped to the same set, result in a high miss count. For each combination, TAC determines its probability and by means of a light-weight cache simulator, the number of misses that would be incurred when the addresses in each combination were mapped to the same set – while the rest of the addresses are randomly mapped. This results in a <*probability*, *misscount*> pair for each combination. The user is then advised to explore random cache placements with the cache simulator until the probabilistic worst-case miss-count (pWCMC) curve derived with EVT eventually upperbounds the pairs determined by TAC. This occurs when enough address combinations (R') singled out by TAC have been simulated and the number of observed miss counts becomes sufficient for EVT to converge to an exponential tail approximation [6]. The user is then instructed to perform R' runs on the actual system to assure a reliable application of MBPTA.

Results with EEBMC Autobench [19] and a railway case study running on a time-randomized FPGA show that TAC successfully identifies conflictive address combinations and determines the number of runs R' required to bring the assurance level of the WCET obtained with MBPTA to a desired threshold.

2 Background

MBTA aims at deriving a WCET estimate that holds during system operation. This requires evidence that measurements taken at analysis occur under conditions similar to or worse than those that can arise during operation. Providing such evidence is out of reach of standard MBTA approaches, as pointed out in Section 1. MBPTA, by deploying EVT (see Figure 1), derives the probability that bad behavior of several of the sources of jitter (*soj*), whose impact has been captured in the analysis-time runs, is simultaneously triggered in the same run, leading to high execution times. Furthermore, randomization makes that *soj* events affecting execution time (including those leading to high execution times) have a probability of appearance. Hence, a probabilistic argument can be built on whether those events are captured in the measurements performed during the analysis phase.

Representativeness defines whether the impact of any random *relevant event* is properly upper-bounded at analysis time. Relevant events are those occurring with a probability above a threshold (e.g. $P_{rel} = 10^{-9}$). With the number of runs R carried out at analysis, only events with a relatively high probability (observable probability or P_{obs}) are (probabilistically) likely to be observed in the measurement runs. This number of runs (R) determines the lowest probability of occurrence of an event such that the probability of not observing it in the analysis time measurements is below a cutoff probability, e.g. 10^{-9}. P_{obs} is a function of the probability of occurrence per run of the event, P_{event}, and the number of runs R (observations) collected by MBPTA at analysis time. For instance, for a cutoff probability of 10^{-9} and $R = 1,000$ runs, we can guarantee that if $P_{event} \geq 0.021$

the event will not be observed with a probability smaller than 10^{-9} (and vice versa). That is, $10^{-9} \geq (1 - 0.021)^{1000}$. It follows that with a higher number of runs, events with lower probability can be captured, though this increases the overhead on the user to deploy MBPTA. *Hence, the relevant events that may not be observed (for $R = 1,000$) with a sufficiently high probability (e.g. $> 10^{-9}$) are those in the range $P_{event} \in [10^{-9}, 0.021]$.*

Benefits and properties of *TRc*: Software complexity in current complex systems is handled via incremental software integration. In the timing domain, caches make the memory layout of existing modules change across integration [16]. This has disruptive effects on time composability since the WCET estimate derived for a software unit in isolation – during system early design stages – is not valid as software integrates. This loss of time composability has potential significant costs since, on every integration, regression tests are required to re-assess the WCET estimate of already-integrated software. Furthermore, timing analysis is pushed and compressed near the end of the development process where the detection of timing violations leads to unacceptable increase in product cost and time to market. *TRc* break the structural dependence among the memory address given to program code/data and its cache set location. The user is not required to control the effect of memory layout but just needs to make sure that its impact on timing has been accounted for performing enough execution time measurements at analysis time. This enables performing measurements *in isolation* factoring in the impact of any cache alignment independently of the memory placement produced by future integration. This has the potential of enabling incremental software integration – and its benefits– in the presence of caches. *TRc* hash addresses with a random number[6] to compute the (random) sets where addresses are placed [11]. The random number remains constant during program execution so that an address is placed in the same set during the whole execution, but it is randomly changed across executions so that the particular set where an address is placed is also random and independent of the placement for the other addresses across executions. Thus, the probability of any two addresses to be placed in the same set is $1/S$ where S is the number of sets.

We call conflictive address combinations, aC_i, those combinations of $W + 1$ or more addresses that, when mapped to the same set, cause a conflictive cache (set) placement that results in a non-negligible increase in execution time. Table 1 summarizes notations used in this paper.

HoG (Heart of Gold) method [3]: *HoG* shows that, whenever up to W addresses are mapped into the same set, after some random evictions, each address can be stored in a different cache line in the set, thus not causing further misses. Conversely, if more than W cache line addresses compete for the cache set space, then they do not fit and evictions will occur often. This scenario represents a random event with high impact on execution time as noticed also in [20, 17]. Hence, a correct application of MBPTA requires ensuring that i) either those events are captured in the measurement runs; or ii) their probability

[6] Random numbers are generated with a pseudo-random number generator that provides sequences with long periods to prevent any correlation.

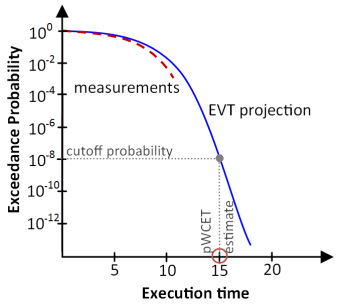

Fig. 1: pWCET (EVT) estimate

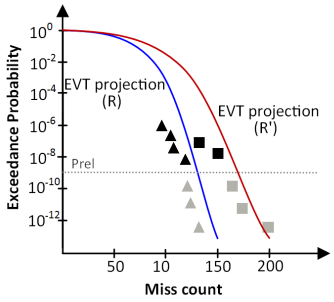

Fig. 2: Application of *TAC*.

is low enough to be considered irrelevant. HoG assumes that *the impact of all addresses on execution time is similar*. This may happen when addresses are accessed homogeneously. However, in the general case not every combination of addresses – when mapped to the same set – results in an execution time increase of the same magnitude. This general case is addressed in this paper.

ReVS (Representativeness Validation by Simulation) method [18]: *ReVS* considers all combinations of the most accessed cache line addresses with a cardinality bigger than W, i.e. $\forall aC_i : |aC_i| > W$, and captures their impact in a cache simulator. However, the number of address combinations with a cardinality bigger than W is huge: $\sum_{k=W+1}^{U} \binom{U}{k}$ for a sequence $\mathcal{Q}_i$, where $U = |@(\mathcal{Q}_i)|$. Hence, evaluating in the cache simulator all potentially conflictive combinations of addresses is *not feasible in the general case* due to its exponential dependence on the number of addresses.

Overall, while MBTA lacks a quantitative measure of coverage of those events that can affect execution time, MBPTA enables deriving a probabilistic argument about whether events impacting execution time are captured in analysis-time tests. Yet, current approaches to derive the number of runs are either non-scalable [18] or assume homogeneous accesses over all program addresses [3]. TAC provides a low-overhead solution to handle the more general case of arbitrary access patterns. For controlled scenarios where ReVS can be applied, e.g. until $U = 15$ addresses, ReVS provides *exact* results with which we compare TAC results to show that TAC covers all conflictive aC_i. We also evaluate TAC in general scenarios, including a real industrial case study.

3 TAC Mechanism

For a sequence of addresses, TAC focuses on identifying *address combinations*, aC_i that, when mapped to the same cache set, cause high execution times. The application of TAC comprises the following steps.

Step 1. List creation. Rather than considering all address combinations with a cardinality bigger than W as ReVS does, TAC provides a list of potential conflictive aC_i ranked according to their expected impact on execution time

Table 1: Definitions used in this paper

Term	Description
aC_i; $\|aC_i\|$	Address combination, i.e. set of unique addresses; cardinality of aC_i
K	Cardinality of (number of addresses in) a combination
$\mathcal{Q}_i$	Sequence of accesses
$@(\mathcal{Q}_i)$; $\|@(\mathcal{Q}_i)\|$	Set of unique addresses in $\mathcal{Q}_i$; Number of unique addresses in $\mathcal{Q}_i$
U	Number of unique addresses in a sequence
X_i	Subsequence of accesses between 2 accesses to the same address
q	Number of distinct addresses in a subsequence X_i
R (R')	Number of measurements to collect determined by MBPTA (TAC)
T	Number of conflictive combinations to return by TAC

(the size of the list is specified later in this section). To that end, TAC builds an *Address Guilt Matrix* (section 3.1) to quickly retrieve those combinations of addresses that, when mapped to the same set, can cause high miss counts.

Step 2. Impact calculation. Each combination in the list is evaluated with a cache simulator. Several Monte-Carlo simulations are performed to derive the number of misses occurring when the addresses in the combination collide in the same set while the rest of the addresses are mapped randomly. The number of combinations in this list is fixed and, therefore, independent of the number of addresses in the program. ReVS, instead, simulates *all* combinations of addresses, which has huge cost.

Step 3. Probability calculation. TAC upper-bounds the probability of occurrence of those aC_i – and combinations of them. The probability of every aC_i to occur is: $S \times (1/S)^{|aC_i|}$, where $|aC_i|$ is the number of addresses in aC_i. For the combined probability of several aC_i we pessimistically use the addition of their individual probabilities. In reality, due to dependences among aC_i, their combined probability is smaller than that [18].

Step 2 and *Step 3* result in a pair <probability, misscount> for each combination. Figure 2 presents a synthetic example where pairs are represented with different symbols: black triangles and squares represent the miss counts obtained for all aC_i – and their combinations – whose probability of occurrence is above P_{rel}. Meanwhile, their gray counterparts are those below P_{rel}, which are discarded by TAC since their probability is deemed as negligible.

Step 4. pWCMC curve. TAC uses MBPTA on the miss counts obtained from cache simulations in which all addresses are randomly mapped, as it would occur in reality, to obtain a probabilistic worst-case miss-count (pWCMC) curve (see solid line in Figure 2). The number of simulations, R, is determined by MBPTA.

Step 5. Assessment. In Figure 2 triangles are those aC_i (and their combinations) whose miss count is covered by the pWCMC, while the miss counts of the aC_i marked with squares are not. Hence, by validating whether the pWCMC curve upper-bounds all conflictive mappings (i.e. <probability, misscount> pairs), we determine whether the number of runs R used by MBPTA suffices. If this is not the case, more runs are performed until the validation step is passed with $R' \geq R$ runs. Whenever it is passed, the number of runs R' is the minimum number of execution time measurements that MBPTA needs to use.

TAC builds on the correlation between miss counts and execution time that has been positively assessed for our target platform in [18]. If such correlation is weak, cache behavior would have low impact in execution time, which would have higher dependence on other *soj*. However, those other *soj* do not challenge MBPTA since probabilities of their events are higher than P_{obs} [3].

3.1 The Address Guilt Matrix

TAC follows an iterative process in which, across iterations, an incremental number of addresses K (starting from $K = W + 1$) is considered to be mapped to the same set. This creates a cache conflict scenario exceeding cache space in one set. The process stops when K is large enough so that the probability of occurrence of the event "K addresses mapped to the same set for the most relevant combinations of K addresses" is below a given cutoff probability[7] P_{cff}. In practice, we only need the most relevant combination for each value of K since EVT (part of MBPTA) already accounts for the probability of several of those events occurring simultaneously. Our results for controlled scenarios show that the worst combination is always among the TAC top-ranked ones, so we consider only the $T = 20$ most relevant combinations for each value of K. In our future work we will investigate how to choose an optimal parameter value for T.

TAC builds on the concept of *guilt*, which is intended to help identifying those aC_i that, if mapped to the same set, result in high miss counts. For a given access A_i with a non-null cache miss probability, guilt provides an approximation to the extent *each intermediate access* between A_i and A_{i-1} causes A_i to miss in cache. Note that this concept, although related, differs from the probability of miss since we are not interested in how many misses each access experiences, but how much certain addresses can impact each other address if placed in the same cache set. For instance, given a direct-mapped (i.e. single way) cache and the sequence $\mathcal{Q}_i = \{A_1B_1A_2\}$, if both addresses A and B are mapped to the same set, A_2 will miss in cache, and the cause of that is access B_1, so B_1 takes full guilt of A eviction. Later in this section we present an efficient mechanism to approximate guilt for arbitrarily complex sequences.

From probability of miss to guilt. Approaches [4] have been proposed to derive upper-bounds to the miss probability. However, in this work we are interested in the *actual* impact rather than on upper-bounds, and on guilt rather than on P_{miss}. Approaches exist to approximate [11] P_{miss} ($\tilde{P}_{miss}$) in the context of MBPTA. These approaches are as shown in Equation 1, where $\sum \tilde{P}_{miss}(X_i)$ corresponds to the accumulated miss probability of the intermediate accesses.

$$\tilde{P}_{miss} = 1 - \left(\frac{W-1}{W}\right)^{\sum \tilde{P}_{miss}(X_i)} \tag{1}$$

While this approach provides good $\tilde{P}_{miss}$ approximations [18], it does not help identifying how much each intermediate access contributes to cause the miss.

[7] Note that, while P_{rel} stands for the threshold probability of relevant events at analysis (e.g., 10^{-9}), P_{cff} relates to the probability of events during operation (e.g., 10^{-15}) [3].

TAC sorts address combinations based on their impact, which requires having means to estimate the relative impact that each address and group of addresses have on each other address (guilt) in terms of cache misses. To cover this gap we propose the $\tilde{P}_{guilty}$ estimator (see Equation 2) that targets providing a precise relative value for guilt as needed by TAC, rather than approximating P_{miss}.

$$\tilde{P}_{guilty} = 1 - \left(\frac{W-1}{W}\right)^{exp} \qquad exp = \begin{cases} 0, & \text{if } q < W \\ q, & \text{if } W \leq q < K \\ K-1, & \text{otherwise} \end{cases} \tag{2}$$

When the number of intermediate addresses between A_i and A_{i-1}, q, is smaller than the number of cache ways W, they all would fit in a cache way, so misses may only be produced due to random replacement, whose impact is already captured with the default number of runs of MBPTA [3]. Hence, we assume that A_i results in a hit, so the guilt of intermediate accesses is 0. Hence, we ignore A_i and look for the next occurrence of A until $q \geq W$ or we reach the end of the sequence. The rationale behind this is that hits do not change cache state in *TRc*, thus they can be ignored. On the other hand, ignoring intermediate accesses due to having extra hits in between A_i and A_{i-1} would be misleading. For instance, let us consider $W = 2$ and $\mathcal{Q}_1 = \{A_1B_1A_2C_1A_3\}$. We cannot assume that A_3 will always hit in $\mathcal{Q}_1$ since sooner or later A will be evicted. Thus, A_2 is ignored and A_3 considers the guilt of B_1 and C_1. It can also be observed that we enforce exp to be smaller than K, the reason behind this is explained next.

Guilt estimation. When for an access A_i $\tilde{P}_{guilty} \neq 0$, its value is 'distributed' among the intermediate accesses between A_i and A_{i-1}. Each access is assigned a guilt value w.r.t. address A computed as shown in Equation 3. For instance given a cache with $W = 2$ ways, the sequence $\mathcal{Q}_1 = (A_1B_1C_1D_1A_2)$ and $K = 3$, we obtain that $q = 3$ and $\tilde{P}_{guilty}(A_2) = 1-(1/2)^2 = 0.75$ according to Equation 2. In this scenario we assign a guilt of 0.375 to each of the $q = 3$ intermediate accesses. Note that the addition of guilt assigned to intermediate accesses is bigger than $\tilde{P}_{guilty}$. The idea is that for $K = 3$, TAC constructs 3-address combinations that in this case can be any of ABC, ABD, ACD, BCD. In all those containing A, we want to assign one half of the guilt to each of the two intermediate accesses. That is, for ABC one half of the guilt is assigned to B and another half to C. At any moment only $K-1$ accesses will be simultaneously considered by TAC, so the guilt of a given access is not decreased because of having other intermediate accesses (more than K). As the value of K increases – as part of TAC iterative process – those other intermediate accesses will be considered simultaneously.

$$guilt = \begin{cases} \frac{\tilde{P}_{guilty}}{exp}, & \text{if } exp > 0 \\ 0, & \text{otherwise} \end{cases} \tag{3}$$

Based on the concept of guilt, which applies at access level, we build the *address guilt matrix* (adgm). The *adgm* comprises as many rows and columns as different (cache line) addresses are accessed in the program. Cell $adgm[A][B]$ captures the guilt of B on A, that is, a measure of to what extent misses of every access A_i are caused by any access to B_j. The *adgm* is built for every value of

K. From the *adgm* we infer information about the impact that each address has on the evictions of each other address. To that end we use the technique in Section 3.2, which covers *Step 1* and *Step 2*. Steps 3 to 5 are applied as presented before.

The metric, obtained from the guilt, does not have a semantic meaning in the real world, yet it provides a way to rank address combinations so that if aCi is ranked higher than aCj, the actual impact in miss count (and execution time) of aCi is higher than that of aCj. This allows performing cache simulations for those highly ranked address combinations to measure their actual impact.

3.2 Smart Search of Address Combinations

Exhaustive Search. As reference we use an algorithm that exhaustively searches the *adgm* and later provide refinements to limit computational costs. For every value of K we build all potential combinations of K addresses out of U, so performing an Exhaustive Search. For each combination we query the *adgm* to obtain the expected impact if those addresses were mapped to the same set. The impact is obtained as follows: (1) for each address i in the combination aCi we compute a value M_i obtained as the highest minimum impact that W other addresses in the combination may have on it. Hence, we take the minimum M_i out of the highest W values in the *adgm* ($adgm[i][x]$ where x is any other address in the combination). Note that we care only about those W addresses that can create highest impact on the address of that row in the *adgm*, since $W + 1$ addresses suffice to exceed the cache set space. Then, we select the minimum value out of those to reflect that, if an address produces few evictions, the others will not produce more evictions than that one because other accesses will become hits. (2) Finally, we obtain the impact as the harmonic mean of all M_i values to, again, reflect that the number of evictions is limited by the address producing the lowest number of evictions. We exclude pairs for the same address (e.g., $adgm[A][A]$) since an address cannot create evictions on itself. If one or some of the addresses have little impact on the other addresses, then its M_i value is much lower and so the final impact, thus allowing to discard this combination. For instance, in the combination $aCi = \{A, B, C, D, E, F\}$, if F has almost zero impact on the other addresses, this combination will be discarded for $K = 6$. If the other 5 addresses have high impact among them, they will be conveniently considered for $K = 5$. Whenever all combinations are considered in the *adgm* (without performing any cache simulation), we create a list of top ranked combinations (*Step 1*) for which cache simulations are performed to measure miss counts (*Step 2*).

Smart Search. Since the computational cost of considering this Exhaustive Search in the *adgm* is prohibitive, we propose a smart search algorithm that comprises the following steps.

First, we discard the rows in the *adgm* whose $\tilde{P}_{guilty}$ is below 1% of the highest $\tilde{P}_{guilty}$ in the table since their combinations with relevant addresses ($\tilde{P}_{guilty}$ above the 1% threshold) will already be accounted by those other addresses, and their impact on irrelevant addresses is deemed irrelevant as well. Then, we create address *buckets* in each row of the *adgm* with all the addresses with the same

guilt value w.r.t. the address of that row. Empirically, we observed that EEMBC and the railway case study produce a low number of buckets. Otherwise, some difference is tolerated among addresses in the same bucket to reduce their count.

Second, the relevant buckets for a certain address are only those whose relative impact w.r.t. the total guilt in the row is significant for the address of that row. Such significance threshold S_{th} (1% in our case) is used to explore combinations with meaningful impact. The remaining addresses (their guilt is below S_{th}) are simply regarded as irrelevant.

Third, we generate the combinations of K elements for each row by making all possible combinations with the address corresponding to that row and $K-1$ elements from different buckets. For instance, assuming $K = 4$ and 2 buckets ($b1$ and $b2$), we make all combinations of 4 addresses using the one of the row and three addresses from the buckets: 3 from $b1$, 2 from $b1$ and 1 from $b2$, 1 from $b1$ and 2 from $b2$, and 3 from $b2$. We always choose those addresses with the highest $\tilde{P}_{guilty}$ in each bucket. We take into account the size of the bucket by computing how many combinations are expected to have the same impact to the representative ones. For instance, if $b1$ and $b2$ contain 4 and 5 addresses respectively, when picking 2 addresses from $b1$ and 1 from $b2$, we determine that there are 30 different combinations meeting those constraints. This is used to set the probability of the pair <probability, misscount> if these combinations have a sufficiently high impact to be simulated.

Fourth, when all addresses have been analyzed and the list with $T = 20$ combinations[8] for a particular value of K is obtained (*Step 1*), we perform cache simulations to determine their miss counts (*Step 2*). In the case of addresses in a bucket, we simulate only those with the highest $\tilde{P}_{guilty}$ and assume the same impact for other combinations that could be generated with other addresses in the bucket. While this may lead to a little pessimism in terms of the impact of those addresses, such pessimism is very limited given that addresses belong to the same bucket. This may result in pairs <probability, misscount> further challenging the reliability of the pWCMC curve, thus potentially rejecting some very tight (yet reliable) pWCMC estimates.

4 Evaluation

We model a pipelined in-order processor with 4KB 2-way-associative 32B-line separated first level instruction (IL1) and data (DL1) caches. Both caches deploy random placement and replacement policies [11], with DL1 implementing write-back (IL1 is read-only). DL1/IL1 access latency is 1 cycle for hits with 3 extra cycles for misses. The latter is added to the main memory latency (16 cycles).

We evaluate TAC on the EEMBC automotive benchmarks, widely used in the community to capture real-time automotive application features [19]. On average this suite has 6,500 Lines of Code, 2,500 Unique Instruction Addresses and 5,600

[8] One combination may be the representative of many others if addresses belong to buckets. Hence, simulating 20 combinations provides information of, at least, 20 actual address combinations, but generally many more than 20.

Unique Data Addresses per benchmark. In particular we use these benchmarks: `a2time` (a2), `aifftr` (at), `aifirf` (ar), `aiifft` (ai), `basefp` (ba), `bitmnp` (bi), `cacheb` (ca), `idctrn` (id) and `iirflt` (ii). We consider all addresses accessed by each benchmark. Additionally, we analyzed the same benchmarks in a *controlled scenario* in which we focus on a subset of the most accessed (cache line) addresses to allow for a comparison against *ReVS*, which hardly scales for higher values of U. While in this scenario we cover on average 58% of the accesses across all benchmarks – thus leaving some degree of uncertainty due to the remaining 42% accesses that are neglected in [18] – it allows comparing TAC against ReVS, with the latter guaranteeing exact results.

TAC vs ReVS. For this comparison we focus only on the U=15 most accessed addresses for which ReVS is capable of exploring all address combinations.

Table 2 shows the number of runs that each of the methods regards as the minimum to use for a reliable MBPTA application. We show results for both IL1 and DL1. As shown, both approaches provide *exactly* the same number of runs (R') for these limited address traces. In particular, TAC identifies the same address combinations most of the times or, alternatively, address combinations with roughly the same impact as those regarded by ReVS as the most conflictive ones for each value of K. The exception to this comes from the case in which ReVS identifies for high values of K combinations which, in fact, are the addition of two or more independent combinations. For instance, ReVS identifies combinations for $K = 6$ that, in reality correspond to two combinations of $K = 3$ occurring at the same time. As explained before, EVT needs to observe high-impact events, but not their combination. Thus, this difference has no influence on R'.

Execution time cost. For $U = 15$ ReVS requires on average 27 hours per benchmark with 1,000 cache simulations per address combination on a cluster running 100 jobs in parallel. TAC is 148 times faster requiring 2 seconds on average per program on a laptop computer to derive the address combinations and their cost, and around 11 minutes per benchmark to run cache simulations for the limited address combinations considered on the same cluster. For full benchmarks, i.e. unrestricted U, ReVS could not be applied while TAC required 1 minute per program to generate the pairs <probability, misscount> and around 38 minutes per program to perform cache simulations in our cluster.

TAC evaluation on full benchmarks. In Table 3 we report the number of runs required by TAC to guarantee that relevant events can only be missed with a probability below a *parametrizable residual threshold*, e.g. 10^{-9}. We also show the runs requested by MBPTA together with the probability of missing those events with the default number of runs required. MBPTA takes as input the number of execution times belonging to the tail of the distribution that need to be observed in measurements, in our case 50 values [2]. Then, starting from 300 runs, MBPTA inspects whether enough tail values are observed. If this is not the case, it asks for more runs until this condition is satisfied and EVT converges.

As shown, $R' \geq R$: in many cases we observe that the likelihood of missing critical address combinations in the default runs (R) determined by MBPTA only is high. This does not mean that pWCET estimates are necessarily wrong,

Table 2: Runs for TAC and ReVS for $P_{rel} = 10^{-9}$ and $U = 15$

	R'_{IL1}		R'_{DL1}		R'	
	ReVS	**TAC**	**ReVS**	**TAC**	**ReVS**	**TAC**
a2	58,360	58,360	540	540	58,360	58,360
at	6,840	6,840	5,500	5,500	6,840	6,840
ar	21,390	21,390	11,530	11,530	21,390	21,390
ai	8,920	8,920	8,770	8,770	8,920	8,920
ba	82,080	82,080	20,010	20,010	82,080	82,080
bi	4,640	4,640	3,510	3,510	4,640	4,640
ca	18,610	18,610	7,950	7,950	18,610	18,610
id	65,770	65,770	47,700	47,700	65,770	65,770
ii	18,310	18,310	49,760	49,760	49,760	49,760

Table 3: Results for complete EEMBC benchmarks

TAC				**MBPTA**	
R'_{IL1}	R'_{DL1}	R'	lik.(R')	R	lik.(R)
67,150	300	67,150	10^{-9}	300	0.911
300	4,760	4,760	10^{-9}	300	0.271
20,080	8,090	20,080	10^{-9}	14,260	10^{-7}
300	10,630	10,630	10^{-9}	300	0.557
78,220	300	78,220	10^{-9}	1,250	0.718
330	1,800	1,800	10^{-9}	300	0.032
19,840	1,500	19,840	10^{-9}	9,360	10^{-5}
67,460	43,040	67,460	10^{-9}	300	0.912
29,920	2,430	29,920	10^{-9}	300	0.812

but indicates that there is non-negligible risk of not observing some high-impact timing events in the analysis runs if TAC is not used.

When comparing the number of runs of TAC with full address traces w.r.t. only 15 addresses, we observe in most of the cases a limited variation in R'. However, in some cases R' decreases noticeably (e.g. R'_{IL1} for `aifftr` (`at`)) because there are many combinations with similar impact that cannot be observed with only 15 addresses. This makes that the probability of observing one of those combinations is much higher and thus, fewer runs are needed to observe one of them. In any case, differently to ReVS, which is limited to 15 addresses, TAC can deal with arbitrary access patterns without any explicit limit. Thus, TAC removes the uncertainty brought by ReVS due to non-analyzed addresses.

5 Railway Case Study

We use as railway case study a safety function part of the European Vital Computer (EVC): the central safety processing unit of the European Train Control System (ETCS) reference architecture. The EVC is responsible of executing all safety functions associated to the travelling speed and distance supervision. As a fail-safe system, whenever an over-speed of the train is detected, the ETCS must switch to a safe-state where the emergency break is active. This safety function shall be provided with the highest integrity level defined in the railway safety standards, SIL-4, and has strict real-time requirements. Accordingly, we apply MBPTA to estimate the WCET for the safety function from the moment of reading the input sensors until the activation of the safe-state. The end user (IK4-IKERLAN) controls input vectors' impact on execution path coverage and in their current timing analysis practice they focus on observed paths. We stick to those paths and apply TAC to all of them. We plan to cover scenarios where the user lacks this control as part of our future work.

Address traces were collected from a LEON3-based FPGA board using existing tracing capabilities of the platform. We have applied TAC to the case study for 10 different input sets (`TEST0` to `TEST9`). The case study comprises around 8,500 Lines of Code, 2,994 Unique Instruction Addresses and 597 Unique Data Addresses for the largest input set.

Table 4: Runs needed by TAC and MBPTA to achieve a confidence of 10^{-9}.

	$IL1$		$DL1$	
	R	R'	R	R'
TEST0	300(Y)	300(Y)	370(N)	1,300(Y)
TEST1	300(N)	600(Y)	3,800(Y)	3,800(Y)
TEST2	300(N)	600(Y)	300(N)	1,000(Y)
TEST3	300(N)	1,600(Y)	300(N)	850(Y)
TEST4	300(N)	1,200(Y)	750(N)	1,100(Y)
TEST5	300(N)	2,100(Y)	480(N)	900(Y)
TEST6	300(N)	500(Y)	890(Y)	890(Y)
TEST7	300(N)	500(Y)	300 (N)	4,400(Y)
TEST8	300(N)	700(Y)	300 (N)	2,300(Y)
TEST9	300(N)	4,800(Y)	1,740(Y)	1,740(Y)

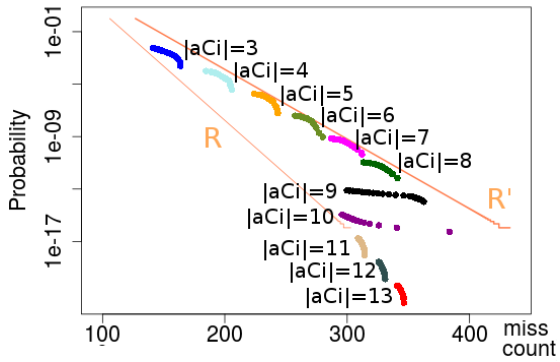

Fig. 3: pWCMC for *TEST7* (DL1) by applying MBPTA (R) and TAC+MBPTA (R').

Table 4 reports the results we obtained, in terms of the number of runs that MBPTA and TAC require in the miss domain. For each test we show whether (Y) or not (N) MBPTA's default number of runs (R) and that reported by TAC (R') suffice to upper-bound the pairs <probability, misscount>. As it can be seen, the default application of MBPTA failed to upper-bound some address combinations for data and instructions for many input sets. Furthermore, in those cases where $R < R'$, confidence on having enough runs for a reliable application of MBPTA cannot be had.

This is illustrated in Figure 3 for TEST7 and the DL1 where TAC <probability, misscount> pairs (points in the plot) are not upper-bounded by the pWCMC curve (lower straight line in the plot) when using $R = 300$, the number or runs required by MBPTA. Instead, if we use $R' = 4,400$, as determined by TAC, the pWCMC curve properly upper-bounds those pairs.

For this industrial application, TAC required, on average, $1,828$ runs per input set, which is affordable in a usual test campaign. TAC took 1.3 minutes to derive the conflictive combinations and 0.35 minutes per test for cache simulations.

6 Related Work

A recent work comparing static (deterministic) timing analysis techniques (SDTA) and MBPTA [1] shows that there is not a dominant technique but the relation between the application working set and the cache size is the factor affecting the most which technique performs better.

MBPTA-compliant hardware. The concept of MBPTA-compliant hardware has been defined in [13]. Hardware techniques provide MBPTA compliance for some specific resources like caches [11] or buses [10]. Software randomization techniques have been shown to enable the analysis of deterministic caches with MBPTA [12]. Time-randomized caches were originally proposed in [11]. Recently some variants have been proposed combining benefits of modulo placement while keeping the randomization required by MBPTA [9]. Some of these random placement designs have been shown to be implementable in FPGA prototypes [9].

Probabilistic Analysis. Some works study random caches in terms of the coverage of conflictive cache placements and complex timing effects, as noted in [3, 17, 20]. Other studies cover aspects related to control-flow dependences and data-dependences in the context of MBPTA. We refer the reader to [13, 23] for details on how to handle control and data dependences.

Applying EVT on software programs brings the dependence of execution times on input-data [23, 15] into the equation. Static and measurement based approaches tackle input-data dependence by requiring program features like loop bounds or recursion level to be bounded to derive WCET estimates. Hence, input vectors mainly affect the paths traversed. Current practice in MBPTA, and our assumption here, is to operate on a set of representative input vectors provided by the user. This is also the practice followed by IK4-IKERLAN for the rail case study. In the context of MBPTA, this assumption can be lifted by synthetically extending the input set, with the same effect of full path coverage [23].

EVT has also been used to estimate WCET on top of non-MBPTA-compliant (deterministic) architectures [7, 8, 5]. The main challenge of those architectures is providing evidence of the representativeness of the execution time observations passed to EVT. To the best of our knowledge, the representativeness challenge has not been studied on non-MBPTA platforms [13].

7 Conclusions

MBTA cannot quantify the degree of coverage attained for the jitter caused by platform events. For caches, while the end user can perform many tests, it is hard to argue about whether conflictive cache mappings leading to high execution times have been covered in the tests. In the context of MBPTA and building on the properties of *TRc*, on every new run a random cache mapping is explored. This enables building a coverage argument. Yet, it is necessary to determine the number of runs to perform to capture conflictive cache mappings. We propose TAC, a low-overhead mechanism that determines whether the number of runs is enough to cover the cache mappings of interest to a given quantifiable threshold.

If this is not the case, TAC requests an increased number of runs to the user until the threshold is reached. Results with EEMBC Automotive and a real railway case study show that TAC successfully identifies conflictive address combinations and increases the number of runs accordingly so that reliable WCET estimates can be obtained for programs with arbitrary access patterns.

Acknowledgments

The research leading to these results has received funding from the European Community's FP7 [FP7/2007-2013] under the PROXIMA Project (www.proxima-project.eu), grant agreement no 611085. This work has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence. Jaume Abella has been partially supported by the Ministry of Economy and Competitiveness under Ramon y Cajal postdoctoral fellowship number RYC-2013-14717.

References

1. Abella, J., Hardy, D., Puaut, I., Quiones, E., Cazorla, F.J.: On the comparison of deterministic and probabilistic wcet estimation techniques. In: 2014 26th Euromicro Conference on Real-Time Systems. pp. 266–275 (July 2014)
2. Abella, J., Padilla, M., del Castillo, J., Cazorla, F.: Measurement-based worst-case execution time estimation using the coefficient of variation. ACM Trans. on Design Automation of Electronic Systems ((to appear))
3. Abella, J., Quiones, E., Wartel, F., Vardanega, T., Cazorla, F.J.: Heart of gold: Making the improbable happen to increase confidence in mbpta. In: 2014 26th Euromicro Conference on Real-Time Systems. pp. 255–265 (July 2014)
4. Altmeyer, S., Davis, R.I.: On the correctness, optimality and precision of static probabilistic timing analysis. In: 2014 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1–6 (March 2014)
5. Bernat, G., Burns, A., Newby, M.: Probabilistic timing analysis: An approach using copulas. J. Embedded Computing 1(2), 179–194 (2005), http://content.iospress.com/articles/journal-of-embedded-computing/jec00014
6. Cucu-Grosjean, L., Santinelli, L., Houston, M., Lo, C., Vardanega, T., Kosmidis, L., Abella, J., Mezzetti, E., Quiones, E., Cazorla, F.J.: Measurement-based probabilistic timing analysis for multi-path programs. In: 2012 24th Euromicro Conference on Real-Time Systems. pp. 91–101 (July 2012)
7. Edgar, S., Burns, A.: Statistical analysis of wcet for scheduling. In: Proceedings 22nd IEEE Real-Time Systems Symposium (RTSS 2001) (Cat. No.01PR1420). pp. 215–224 (Dec 2001)
8. Hansen, J.P., Hissam, S.A., Moreno, G.A.: Statistical-based WCET estimation and validation. In: Holsti, N. (ed.) 9th Intl. Workshop on Worst-Case Execution Time Analysis, WCET 2009, Dublin, Ireland, July 1-3, 2009. OASICS, vol. 10. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, Germany (2009), http://drops.dagstuhl.de/opus/volltexte/2009/2291
9. Hernandez, C., Abella, J., Gianarro, A., Andersson, J., Cazorla, F.J.: Random modulo: A new processor cache design for real-time critical systems. In: 2016 53nd ACM/EDAC/IEEE Design Automation Conference (DAC). pp. 1–6 (June 2016)

10. Jalle, J., Kosmidis, L., Abella, J., Quiones, E., Cazorla, F.J.: Bus designs for time-probabilistic multicore processors. In: 2014 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 1–6 (March 2014)
11. Kosmidis, L., Abella, J., Quiones, E., Cazorla, F.J.: A cache design for probabilistically analysable real-time systems. In: 2013 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 513–518 (March 2013)
12. Kosmidis, L., Curtsinger, C., Quiones, E., Abella, J., Berger, E., Cazorla, F.J.: Probabilistic timing analysis on conventional cache designs. In: 2013 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 603–606 (March 2013)
13. Kosmidis, L., Quiones, E., Abella, J., Vardanega, T., Broster, I., Cazorla, F.J.: Measurement-based probabilistic timing analysis and its impact on processor architecture. In: 2014 17th Euromicro Conference on Digital System Design. pp. 401–410 (Aug 2014)
14. Kotz, S., Nadarajah, S.: Extreme Value Distributions: Theory and Applications. EBL-Schweitzer, Imperial College Press (2000), https://books.google.es/books?id=tKlgDQAAQBAJ
15. Lima, G., Dias, D., Barros, E.: Extreme value theory for estimating task execution time bounds: A careful look. In: 2016 28th Euromicro Conference on Real-Time Systems (ECRTS). pp. 200–211 (July 2016)
16. Mezzetti, E., Vardanega, T.: A rapid cache-aware procedure positioning optimization to favor incremental development. In: 2013 IEEE 19th Real-Time and Embedded Technology and Applications Symposium (RTAS). pp. 107–116 (April 2013)
17. Mezzetti, E., Ziccardi, M., Vardanega, T., Abella, J., Quiones, E., Cazorla, F.: Randomized caches can be pretty useful to hard real-time systems. Leibniz Transactions on Embedded Systems 2(1), 01–1–01:10 (2015), http://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v002-i001-a001
18. Milutinovic, S., Abella, J., Cazorla, F.J.: Modelling probabilistic cache representativeness in the presence of arbitrary access patterns. In: 2016 IEEE 19th International Symposium on Real-Time Distributed Computing (ISORC). pp. 142–149 (May 2016)
19. Poovey, J.A., Conte, T.M., Levy, M., Gal-On, S.: A benchmark characterization of the eembc benchmark suite. IEEE Micro 29(5), 18–29 (Sep 2009), http://dx.doi.org/10.1109/MM.2009.74
20. Reineke, J.: Randomized caches considered harmful in hard real-time systems. Leibniz Transactions on Embedded Systems 1(1), 03–1–03:13 (2014), http://ojs.dagstuhl.de/index.php/lites/article/view/LITES-v001-i001-a003
21. Wartel, F., Kosmidis, L., Gogonel, A., Baldovino, A., Stephenson, Z., Triquet, B., Quiones, E., Lo, C., Mezzetta, E., Broster, I., Abella, J., Cucu-Grosjean, L., Vardanega, T., Cazorla, F.J.: Timing analysis of an avionics case study on complex hardware/software platforms. In: 2015 Design, Automation Test in Europe Conference Exhibition (DATE). pp. 397–402 (March 2015)
22. Wilhelm, R., Engblom, J., Ermedahl, A., Holsti, N., Thesing, S., Whalley, D., Bernat, G., Ferdinand, C., Heckmann, R., Mitra, T., Mueller, F., Puaut, I., Puschner, P., Staschulat, J., Stenström, P.: The worst-case execution-time problem—overview of methods and survey of tools. ACM Trans. Embed. Comput. Syst. 7(3), 36:1–36:53 (May 2008), http://doi.acm.org/10.1145/1347375.1347389
23. Ziccardi, M., Mezzetti, E., Vardanega, T., Abella, J., Cazorla, F.J.: Epc: Extended path coverage for measurement-based probabilistic timing analysis. In: 2015 IEEE Real-Time Systems Symposium. pp. 338–349 (Dec 2015)