Increasing the Reliability of Software Timing Analysis for Cache-Based Processors

Suzana Milutinovic, Enrico Mezzetti, Jaume Abella, and Francisco J. Cazorla
Barcelona Supercomputing Center (BSC), Spain

I. Introduction

In critical-embedded real-time systems [34] software continues to be in charge of providing most innovative services, making it instrumental in increasing products' competitive edge in the market [23]. Software is also increasingly driving the decision making process over a huge amounts of data of diverse types, which not only increases its complexity but also complicates timing validation and verification (V&V). The latter focus on providing evidence that system functions perform timely: to that end, timing analysis methods are used to estimate the worst-case execution time (WCET) of tasks. WCET estimates must be reliable, according to the level of confidence defined in the relevant safety standards, and as tight as possible, to minimize the provisioning of hardware resources. Timing V&V is further challenged by the use of performance-accelerating hardware (e.g., caches) to provide the unprecedented rise in performance needs for critical software's, expected to be as high as 100x in the next years in the automotive domain [1].

Increased hardware and software complexity reduces the confidence that can be placed on WCET estimates derived by measurement-based timing analysis (MBTA), the most used timing analysis technique in critical real-time embedded systems [44]. In particular, increased effort is needed on the user to concoct stressing execution scenarios during the (analysis-time) test campaign as a means to capture bad scenarios that can arise during system operation [18].

Complex hardware/software platforms exacerbate the inherent variability in the execution time of a program, leading to timing distributions with arbitrary variance and shape. This has motivated the use of statistical techniques to derive bounds to execution time distributions. In particular Measurement-based probabilistic timing analysis (MBPTA) [3], [8], matured in recent years, delivers a probabilistic WCET (pWCET) function that upper-bounds the (probabilistic) execution time distribution of the program (pETd) at any exceedance probability, see Figure 2. MBPTA has been successfully applied to industrial case studies [43] and its impact on certification has been addressed [42].

MBPTA has been complemented with solutions that inject randomization in program's timing behavior to relieve the user from controlling those *jittery resources* affecting the execution time variability of a program. Randomization makes that all potential behaviors that a given jittery resource (e.g. caches) can exhibit, are naturally (and randomly) explored in every new test, enabling the derivation of probabilistic guarantees. In the case of caches, cache-placement randomisation breaks the dependence between memory mapping and cache location, typical of conventional modulo-placement caches. As will be detailed later, this prevents incremental software integration from having any repercussion on cache behavior, thus not requiring the end user to exercise any control over memory mapping [12]. Randomization has been implemented at hardware level (e.g. random arbitration policies and random placement/replacement techniques) that are now part of a commercial product for the space domain [6]; and with software-only techniques (e.g. compiler level) [26]. We focus on hardware-randomized cache placement.

To properly account for the timing behavior of caches – one of the on-chip processor resources with the highest impact on average and worst-case performance – timing V&V requires capturing Conflictive Cache Placements (CCP) during the test campaign. Under a CCP the number of addresses mapped to a cache set exceeds its associativity W. Having $W + 1$ or more addresses mapped to the same cache set will cause a notable increase in the number of misses and, eventually, in the execution time [2]. A concern for timing V&V arises when the CCP occur with a sufficiently high probability, deemed relevant by the corresponding safety standard, but low enough not to be observed in the measurements at analysis time [2], [31], [39]. For instance, for a program accessing 5 addresses, the probability that all of them are randomly mapped to the same set in a 32-set 4-way cache is $10^{-6} \approx 32/32^5 = (1/32)^4$, which is considered to be relevant for avionics and automotive products[1]. With random cache placement, when $R = 1,000$ runs are performed – a reference value typically used by MBPTA – the probability that at least one run captures the execution-time impact of the cache placement of interest (i.e., five addresses mapped to the same set) is very low ($\approx 10^{-3}$), and hence, highly unlikely to be captured by MBPTA. Thus, MBPTA faces the challenge of deriving an R guaranteeing that relevant CCPs of time-randomized caches (TRc) are captured with sufficiently high probability.

High values of R increase the probabilistic coverage of CCPs, but also increase the application costs of MBPTA, drastically reducing its overall benefit/cost ratio. In this paper, we provide means to quantify the probability (risk) of not capturing CCPs, and the increased cost of performing more

[1]Depending on the criticality level, acceptable per-hour failure rate probabilities range from 10^{-6} to 10^{-9}.

runs. This provides the system engineer a mechanism to take an informed decision on the number of measurements to collect, properly balancing the time/effort available for the analysis, the criticality of the software being analyzed, and the corresponding safety requirements in the reference application domain.

We focus on the two TRc existing designs, namely, Random Modulo (RM), already implemented on a commercially-available processor [6] with competitive results in comparison to standard (modulo placement) caches [22]; and hash-based Random Placement[2] (hRP) that provides lower performance though it imposes fewer constraints in hardware designs, thus offering a different and valuable tradeoff [27]. In particular, the main contributions of this work are as follows:

We analyze RM and hRP to shed some light on how CCPs emerge in a different manner under those designs. We show that the particular set of $W + 1$ or more addresses that when mapped to the same set cause a CCP are different under RM and hRP. Further, both the miss counts and the number of CCP decrease under RM, which in turn reduces the number of runs required. This motivates the necessity of different design-specific techniques to intercept CCPs.

We propose CCP-RM and CCP-hRP that identify the CCP for a given sequence of addresses, along with their probability and impact on execution time, for RM and hRP respectively. For a given configurable coverage probability threshold P_{cth}, CCP-RM and CCP-hRP determine whether CCP's impact is captured in the default R runs performed by MBPTA. Otherwise, more runs need to be carried out until the probability of not observing one of the random CCP is below P_{cth}.

We evaluate CCP-RM and CCP-hRP on a cycle-accurate timing simulator where we provide evidence of their benefits on reference EEMBC automotive benchmarks. The simulator experimental setup allows building controlled scenarios in which we can exhaustively derive *all* cache placements and show how CCP-RM (CCP-hRP) capture the actual set of conflictive cache placements. This information can be used as evidence for certification.

We assess CCP-RM and CCP-hRP on a real setup with a case study, representative of the railway domain, running on real implementations of RM and hRP on an FPGA board. Results show that CCP-RM and CCP-hRP identify CCP with limited burden on the user side (in terms of effort and time) and derive the number of observations R' that needs to be collected to ensure that the probability of not capturing a relevant CCP is below acceptable levels.

The rest of this paper is organized as follows. Section II presents basic background on timing Validation and Verification. Section III shows the main differences in terms of CCP between hRP and RM. Sections IV and V detail our CCP-RM and CCP-hRP proposals respectively, which are then thoroughly evaluated in Sections VI and VII, on simulated and FPGA platforms respectively. Finally, Section VIII covers the most relevant related works whereas Section IX draws the conclusions we derive from this work.

[2]Whereas the CCP-hRP mechanism and its evaluation have been already published in [33], the CCP-RM mechanism, the analysis of RM and hRP caches and the comparison of CCP-RM and CCP-hRP are strictly novel contributions.

II. Timing V&V, Certification, and Standards

Timing Faults and Safety Standards. A common misconception in real-time systems is that a program overrun (timing fault) necessarily causes a failure at system level. In reality, a safety process factors in the impact that timing faults can have on the overall system failure rate. Taking as a reference the ISO26262 [24] standard in the automotive domain, the safety life cycle defines safety goals (and their associated Automotive Safety Integrity Level or ASIL) for each system element. If those goals are reached, the *residual risk* of failure is deemed as sufficiently low. The safety process also defines the safety requirements on the hardware/software to reach the safety goal. Proper measures are put in place to reduce the probability that a fault in a hardware/software element can contribute to the violation of its safety requirements – and hence the safety goal – beyond an established threshold. For instance, random hardware residual faults are considered acceptable if their rate is below a given threshold and the diagnosis coverage – i.e. the mechanisms detecting whether this type of fault can occur for a given hardware block – is below a given target. For instance, for the highest safety level (ASIL-D) the maximum allowed failure rate is 10^{-8} per hour of operation when the diagnosis coverage reaches 99%.

MBTA and evidence for certification. For complex hardware and software, industry will continue to use MBTA as the main analysis approach. This has not only been directly acknowledged by automotive representatives [38], but also, in recent industrial works, OEM/Tier1 teams and static timing analysis (STA) tool providers increasingly resort to measurement-based analysis to derive timing bounds for processor architectures like the NXP P4080 [14], and ARM-based SABRE Lite multicore system [11]. The reason is that STA does not scale to handle the complexity of hardware and software: STA aims at delivering evidence for timing V&V models building on formal proofs that are meant to show the soundness of the application of the analysis steps. While this can be agreed to be scientifically sound, the confidence on STA results still builds on the fragile and increasingly unrealistic assumption of correctness of the underlying timing model, which is expected to faithfully represent the behavior of the real hardware being analyzed. The extraordinary complexity of multicore platforms and the lack of sufficient technical details in manuals cause the above assumption to be poorly sustainable in practice, which in turn drives the industrial practice to build on empirical evidence (measurements) as the main element to show adherence to timing requirements. In this line, most recent industrial approaches to handle the timing of complex hardware [5] build on requirements that, for the same reasons above, can only be assessed empirically. This includes, for instance, the identification of interference channels, which necessarily requires an extensive set of tests

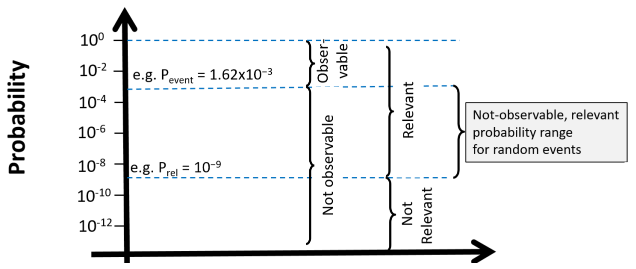


Fig. 1. Probability range of interest

showing the processor resources in which tasks affect each other's timing behavior.

Furthermore, *empirical evidence and heuristics are widely used in hardware testing processes*. For instance, ISO26262 defines different failure rate thresholds for different ASIL levels according to the diagnostic coverage w.r.t. residual faults (see clause 9.4.3.6 in ISO26262 Part 5). Fault models used for assessing diagnostic coverage are in many cases restricted to models that can be tested at appropriate abstraction levels. For instance, stuck-at faults can be easily assessed at gate-level, where the model of the hardware is precise enough, and engineers have full controllability and observability of the circuit. In the absence of 'complete' fault models to assess diagnostic coverage, heuristics are used to generate a sufficiently low number of tests with sufficiently high diagnostic coverage. For instance, tests are normally generated with Automatic Test Pattern Generators (ATPG) [7], [16], which build upon heuristics (including guided search algorithms). Target test coverage can be as low as 90% even for the highest criticality levels (ASIL-D in automotive), thus leaving up to 10% of the design untested against relevant faults. Overall, evidence for certification builds upon measurements obtained with heuristics due to the inability to afford exhaustive explorations.

MBPTA. The quality of WCET estimates is hard to assess, especially for increasingly complex systems. The main concern in MBTA lies in the construction of tests cases for the test campaign that exhaustively (and simultaneously) capture the worst-case behavior of jittery resources (*jres*). MBPTA applies to timing analysis the ISO26262's philosophy to handle hardware faults: MBPTA derives tasks' probabilistic distributions, representing the probability that one task execution overruns a given budget, as shown in Figure 2. For example, assuming a budget of 7 time units, the probability that a task exceeds its deadline (potentially causing a timing failure) is below 10^{-10}. By multiplying the target probability by the frequency of execution of the analyzed task per hour, MBPTA derives the probability of timing failures per hour of operation, hence fitting ISO26262.

MBPTA controls the impact on execution time of *jres* during the test campaign. It does so by either enforcing *jres* that cause low execution-time variation to work on their worst latency (so their impact on execution time is upperbounded in analysis-time runs) or by randomizing the timing behavior of those *jres* that cause high execution-time variation (such that bad timing behavior is captured with a quantifiable increasing probability as more measurements are taken). As a result, the analysis time pETd (ApETd) is an upperbound of the operation time pETd (OpETd) (see Figure 2). On the other hand, MBPTA simplifies the process of collecting observations during the test campaign by deploying statistical techniques such as Extreme Value Theory [29] (EVT). EVT models program's execution time probability distribution based on a limited number of measurements (e.g. 1000), see pWCET distribution in Figure 2. Further, EVT transparently derives the combined probability that the bad behavior of different *jres* occur simultaneously in a single run, provided that the bad behavior of each single *jres* has been captured in the collected runs. As a consequence, triggering the bad timing of all *jres individually* suffices for a trustworthy WCET estimation with MBPTA. This is in contrast to MBTA that requires all bad behavior to be triggered in a single run.

The quality of WCET estimates obtained with MBPTA depends on the *representativeness* of the measurements collected at analysis w.r.t. the timing behavior of the system during operation. In analysis-time measurements, bad timing behavior of each *jres* needs to be exercised and fed to EVT, which is then responsible for modelling the combined behavior of several *jres*. For low-variability *jres*, which are enforced to work at their highest latency at analysis, a single measurement is sufficient to capture bad behavior. For high-variance *jres*, which are instead meant to be time-randomized, all *relevant (random) events* need to be captured by carrying out *enough runs*.

Relevant (random) events, see Figure 1, are all those occurring with a probability above a threshold that relates to the corresponding safety standard in the domain (e.g. $P_{rel} = 10^{-9}$). The probability of not observing a relevant random event at analysis (P_{nobs}) is a function of the number of measurements taken (R) and the event probability ($P_{nobs} = (1 - P_{event})^R$). See Table I for the definitions used throughout the paper. For example, let us assume that R = 10,000 measurements are taken during the test campaign at analysis. The events with per-run probability of occurrence equal to P_{event} may be not be properly covered (i.e. missed) with negligible probability, P_{cth}, if $(1 - P_{event})^{10000} \leq P_{cth}$. For instance, for $P_{cth} = 10^{-7}$ events with a probability $P_{event} \geq 0.00162$ are captured with a probability higher than $1 - 10^{-7}$. To capture events with lower probabilities of occurrence with enough confidence, more runs are needed.

III. Understanding CCP under RM and hRP

For RM and hRP, the relevant random events to track are the CCPs, corresponding to those random cache mappings that cause the number of program addresses mapped to the same set to exceed the cache associativity W [2]. Those addresses compete for the same set, increasing the number of conflict misses. Otherwise, i.e. when W or fewer addresses within a loop are mapped to the same set, after some initial misses due to random replacement, the addresses will eventually end

TABLE I
DEFINITIONS USED IN THIS PAPER.

Term	Description
P_{event}; P_{rel}	Probability of ‘event’ (e.g.CCP); Threshold probability separating relevant from non-relevant events
P_{nobs}; P_{cth}	Probability of missing event at analysis; Coverage threshold probability dictated by safety standard
$\tilde{P}_{guilty}$	Probability of the address of interest to be evicted
$guilt$	Measure of the contribution of a given address to the $\tilde{P}_{guilty}$ of the address of interest
K	Cardinality of (number of addresses in) a combination
$\mathcal{Q}_i = \{A_j^X, ...\}$	Sequence of accesses, A is the memory addresses, j the number of times A has been referenced in the sequence and X is the cache segment A belongs to
$@(\mathcal{Q}_i)$; $\lvert @(\mathcal{Q}_i)\rvert$	Set of unique addresses in $\mathcal{Q}_i$ and Number of unique addresses in it
memaddr^X	Any of the accesses in sequence to memory address *memaddr* belonging to the segment X
U	Number of unique addresses in a sequence
X_i	Sequence of accesses between 2 accesses to the same address
q	No. of distinct addresses in a sequence X_i
s	No. of distinct cache segments in a sequence X_i
$m1(m2)$	Local variables used to compute $P_{guilt-seg}$ $(guilt)$
R $(R'$ $)$	Number of measurements to collect determined by MBPTA (CCP-aware mechanism: CCP-RM/CCP-hRP)
T	Number of conflictive combinations
S; W	Number of cache sets; Number of cache ways

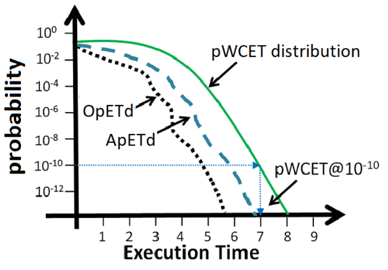


Fig. 2. Randomization (and upperbounding) of jittery resources cause the analysis time pETd (ApETd) to upperbound that at operation (OpETd). Building on a sample (of for instance 1,000 elements) from ApETd, MBPTA derives a probabilistic WCET distribution.

up fitting in their assigned cache set [2]. We define two distinct procedures, CCP-RM and CCP-RP, that derive the CCP probability (i.e. P_{event}) on RM and hRP caches respectively; they also determine whether the probability (P_{nobs}) of not observing CCP with the default number of measurements R is below a given user-provided coverage threshold (P_{cth}). If $P_{nobs} > P_{cth}$, CCP-RM and CCP-hRP request the user to perform more runs R' until $P_{nobs} < P_{cth}$, i.e. the probability of not capturing CCP is below the user-defined threshold. Hence, CCP-RM and CCP-hRP offer the user a mechanism to balance testing time and providing evidence for certification on timing budgets according to the target integrity (criticality) level.

A. Time-Randomized Cache (TRc) Implementations

TRc breaks the dependence between address location in memory and its cache set position. As a result, during the test campaign, users do not need to control program's code/data memory placement – which is very sensitive to environmental execution conditions that may change across software integration steps – but only need to adjust the number of measurements to perform. Further, TRc favor *incremental software integration* in that, unlike traditional caches, TRc cause that changes in the memory layout of a module, occurring when new modules are integrated, do *not* affect the set of CCPs. Hence, while with traditional caches new memory layouts invalidate the WCET estimates previously derived under a different layout (software integration) [12], this is not the case for TRc.

Hash-Based Random Placement (hRP) hashes addresses with a random number to derive the cache set in which to place the address. The random number is changed at the beginning of each execution, such that addresses change placement across sets. Under hRP any two addresses[3] can be mapped to the same set with a probability $1/S$ [$=S/S^2$], where S is the number of sets. The main disadvantage of hRP is that it exhibits cache conflicts even in the scenarios where all data (or the subset of most-accessed data) is largely below cache capacity since all addresses can be potentially placed in the same set.

Random Modulo placement (RM) groups the addresses sharing the same memory page into *cache segments* and ensures that all addresses from the same segment are mapped to distinct sets. Randomization is achieved by using the random number (changed across executions) and some address bits to drive the permutation of index bits to the bits denoting the set where to map the address. By avoiding conflicts among consecutive memory addresses (those in the same page), RM better exploits spatial locality in a similar manner as modulo placement, and outperforms hRP in terms of both average and worst-case performance. Note that RM requires that page alignment does not change upon integration. Otherwise, WCET estimates obtained under the assumption that some contents reside in the same page (and hence cannot conflict in cache), would be no longer valid if those contents move to separate pages. Thus, RM improves performance w.r.t. hRP, but also poses some additional constraints. RM preserves the properties needed by MBPTA (independence of actual memory addresses) as long as the page size is equal (or a multiple) of the way size. If the page is smaller, then hRP must be implemented instead of RM.

B. State-of-the-Art on CCP Detection

Cache placement plays a key role in the amount of conflict misses experienced, becoming a central element for several average- and worst-case cache optimization approaches [17], [21], [30], [36]. These approaches, whose focus is on modulo placement, build on deriving the number of cache conflicts by either statically analyzing or profiling a program under a heuristically determined subset of cache placements. This information is later used to select the cache placement (among those observed) that produces the minimum number of conflicts. These approaches, however, build on the unlikely assumption that a cache layout is robust against program

[3]For the sake of simplicity we assume that the addressable unit matches the size of a cache line.

variations and incremental system integration, widely spread in automotive and avionics among other domains. Our interest in CCPs is not in finding an optimal one (which is indeed a fragile concept) but rather in providing guarantees that relevant CCPs are covered in the analysis test campaign.

The need to identify CCPs, and their probability of occurrence, in the scope of MBPTA was first shown in [2], which propose a mechanism for hRP that determines the number of measurements R to carry out as a function of the number of program addresses and the probabilities that more than W of them reside in the same set. Such approach assumes private and partitioned caches, which are normally deployed in critical real-time systems, for the sake of time-predictability. However, the proposed approach is only suitable when all addresses have a similar impact on execution time, while in general access patterns and access counts across different addresses are arbitrary.

ReVS [32] is a theoretical evaluation framework proposed for hRP, but that can be adapted for all TRc. It builds on Monte-Carlo simulations to explore the impact of a relatively large set of random cache placements. From these simulations the authors evaluate the miss impact in a cache simulator of *all* potential CCPs. ReVS time overheads are unaffordable in the general case as the number of cache placements to be considered is a function of the number of different (unique) addresses in the program: with address counts as small as 20 the number of simulations to perform already reaches the order of billions, dismissing ReVS as a valid general solution. Instead, ReVS can be used in controlled experiments with reduced number of addresses, to assess the accuracy of computationally-tractable techniques (as CCP-RM and CCP-hRP) to capture CCP.

To the best of our knowledge, (1) no technique allows deriving the minimum number of runs required for hRP with affordable computational cost, and (2) no solution has been proposed for higher-performance RM caches.

C. Difference between RM and hRP

RM and hRP cause CCP to be triggered differently, which motivates having two different CCP detection techniques. As an illustrative example, Figure 3 shows how 4 addresses could be mapped to a 4-set cache in different runs. We see that, since the addresses belong to the same memory page, they cannot be mapped to the same set under RM and hence, cannot incur CCP. Instead, with hRP every single address can be mapped to any set, and possibly trigger CCP.

In order to better understand the differences between RM and hRP let us assume the address sequence $\mathcal{Q}_0$.

$$\mathcal{Q}_0 = \{A_1^0 B_1^0 C_1^1 D_1^1 C_2^1 D_2^1 C_3^1 D_3^1 A_2^0 B_2^0 \\ A_3^0 B_3^0 C_4^1 D_4^1 C_5^1 D_5^1 C_6^1 D_6^1 A_4^0 B_4^0\}$$

Each element is a memory address with the superscript denoting the memory page (cache segment) it belongs to, and the subscript the number of times that address has been referenced. Further let us assume a 2-set direct-mapped cache. With hRP, addresses have equal probability to be mapped to

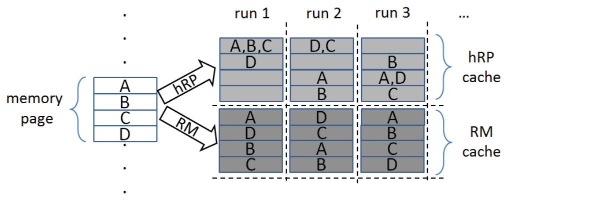


Fig. 3. Small example for a 4-way hRP and RM cache. For RM the CCP arises across different memory pages.

a set. The same holds for RM with the exception of addresses belonging to the same memory page that are prevented from colliding in the same set. Consequently, the set of possible cache placements generated by RM is always a subset of the possible cache placements by hRP. For $\mathcal{Q}_0$, Table II lists all possible placements. As it can be seen, among all hRP placements, only two can arise with RM.

TABLE II
CONFLICTIVE CACHE PLACEMENTS (CCP) FOR $\mathcal{Q}$ UNDER RM AND HRP

Placement	hRP		RM	
{set0}{set1} or {set1}{set0}	Misses	CCP?	Misses	CCP?
{ABCD} {-}	20	yes	not possible	
{AB} {CD}	20	yes	not possible	
{BCD} {A}	17	yes	not possible	
{ACD} {B}	17	yes	not possible	
{ABC} {D}	15	yes	not possible	
{ABD} {C}	15	yes	not possible	
{AC} {BD}	10	yes	10	yes
{AD} {BC}	10	yes	10	yes

hRP and RM produce different CCPs. hRP does not prevent any addresses to coexist in a set, resulting in potential conflicts among addresses in the same memory page. RM placement, instead, avoids such behavior by design. Moreover, the CCPs with RM, in general, produce lower miss counts compared to those of hRP, as illustrated in Figure 4, reporting miss count and frequency for the `bitmnp` EEMBC benchmark executed in our reference setup (later presented in Section VI). RM incurs similar miss counts as hRP only when addresses of the sequence causing conflict misses belong to different memory pages.

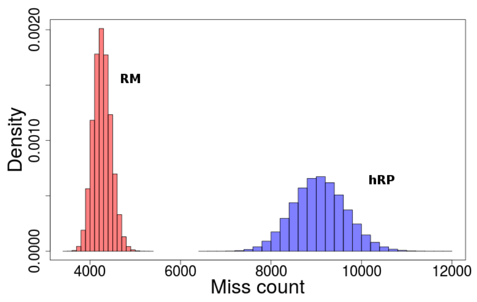


Fig. 4. `bitmnp` behavior in first level instruction hRP and RM caches.

IV. The CCP-RM mechanism

CCP-RM derives the minimum number of runs needed to ensure that all relevant cache events (i.e. CCP) have been observed with sufficiently high probability for an RM cache. For a given cache setup, CCP-RM analyzes the sequence of memory accesses of the program (considering instruction and data accesses separately). From these inputs, CCP-RM:

1) first creates a list of those address combinations that, if placed in the same set, can result in high cache miss counts (Section IV-A).
2) derives for each combination its impact in miss count and probability of occurrence (Section IV-B).
3) assesses whether for a probability threshold $1 - P_{cth}$ the number of runs typically required by MBPTA (R) already captures these combinations and, otherwise, iteratively requests further runs (R') until those combinations are not observed with a maximum probability below P_{cth} (Section IV-C).

CCP-RM is heuristic centric to make it computationally tractable. Our results show that, in practice, the actual CCPs are captured in the top 3 address combinations identified by CCP-RM. Further, those address combinations classified as the top ones by CCP-RM, but which do not correspond to the actual CCP, have in fact, similar impact to the actual CCP.

A. Deriving Relevant Address Combinations

A necessary characteristic of CCP address combinations is that their cardinality (i.e. number of addresses) exceeds the number of cache ways W (e.g. for a 2-way cache, all address combinations of 3 or more addresses are potentially conflictive). CCP must also have a probability of occurrence sufficiently high to be considered relevant by the corresponding safety standard.

For each address count $K > W$, CCP-RM derives a list of addresses (combinations) expected to generate many misses if they are randomly mapped to the same set. CCP-RM focuses on those combinations whose probability of occurrence is above a safety-standard defined probability (e.g. 10^{-9}). It stops exploring K values when the probability that K addresses are mapped to the same set falls below that threshold. Such probability is analytically derived as $(1/S)^{K-1}$, (C3:) the multiplication of the number of available sets S and individual probabilities to be mapped to the selected set $1/S$, for each of the K addresses). CCP-RM sets a maximum size (T) for the list of most relevant combinations to be considered for a given value of K. As detailed later, those T combinations, once evaluated, provide information of at least T address combinations but, due to the way we select them, often represent many more than T combinations.

Next, we describe how to produce the lists (for each value of K). We accompany the explanation with examples in which we assume a 2-way set-associative cache.

1) Guilt Estimation: We introduce an estimator (loosely based on the probabilities of address misses) called *guilt* that, for each address A^X in a program, classifies the other addresses with respect to how many evictions of A^X they will cause if randomly mapped to the same set. The *guilt* attribute is later exploited to calculate the predicted impact of a group of addresses, which in turn is used to rank the address combinations based on the increase of the overall number of cache misses caused by these addresses placed together in the set. To estimate guilt, CCP-RM analyzes the access sequence between consecutive accesses to the same address.

Notably, under RM only addresses belonging to different segments can evict each other. Hence, CCP-RM ignores addresses belonging to the same segment. For example, in the sequence $\mathcal{Q}_1 = (A_1^{100}, B_1^{100}, C_1^{100}, A_2^{100})$, A_2^{100} is necessarily a hit, since all intermediate accesses, i.e. those between A_1^{100} and A_2^{100}, belong to the same segment.

Also with RM, addresses mapped to a set necessarily belong to different segments. Therefore, an address can only conflict with *exactly one* address from every other segment, but all addresses from that segment are equally probable to conflict. For example, in $\mathcal{Q}_2 = (A_1^{100}, B_1^{102}, C_1^{102}, A_2^{100})$, addresses B^{102} and C^{102} both can be mapped with the same probability $1/S$ to the same set as A^{100}, but only one of them will be actually mapped (i.e. either B^{102} or C^{102}). Hence, the number of addresses accessed in-between two accesses to the considered address (A^{100}) that can be placed in the same set is at most the number of different cache segments accessed in between, denoted by s.

For each memory access A_j^X, we define $\tilde{P}_{guilty}$ as the likelihood of A^X being evicted due to the conflicts with other addresses, see Equation 1. Since in RM caches, conflicts can only happen between addresses mapped to different cache segments, s in Equation 1 represents the number of distinct cache segments accessed in between two accesses (A_{j-1}^X and A_j^X) to the same address A^X.

$$\tilde{P}_{guilty}(A_j^X) = 1 - \left(\frac{W-1}{W}\right)^{m_1} \quad m_1 = \begin{cases} 0, & \text{if } s < W \\ s, & \text{if } W \le s < K \\ K-1, & \text{otherwise} \end{cases} \tag{1}$$

The fraction $\frac{W-1}{W}$ represents the probability of a cache line to survive an eviction, whereas m_1 relates to the number of evictions occurring. When s is smaller than W, the intermediate accesses would fit in a cache set, so misses may only be produced due to random replacement, whose impact is already captured with the default number of runs of MBPTA [2]. Hence, we assume that A_j^X hits, so the guilt of intermediate accesses is 0. For values of s larger or equal to W, we assume s evictions, but bounding that number up to $K - 1$ since we inspect different address group cardinalities (K) iteratively and for a given K value at most $K - 1$ addresses can conflict with the address in focus.

Equation 1 captures the case when the number of distinct segments accessed between A_{j-1}^X and A_j^X is higher or equal to W (i.e., $s \ge W$). Conflicts can also occur under other address interleavings. For example, in the sequence $\mathcal{Q}_3 = (A_1^{100}, B_1^{101}, A_2^{100}, C_1^{102}, A_3^{100})$, A_3^{100} is likely to suffer misses, even though the estimator would predict hits because

$s < W$. In this case, we may have misses because hits do not alter cache state[4] so that they can be stripped out of the access sequence conceptually (e.g. A_2^{100})), thus making other accesses (e.g. A_3^{100}) likely miss due to evictions caused by B_1^{101} and C_1^{102}. To account for this, when deriving $\tilde{P}_{guilty}$ of an access, CCP-RM searches for the previous access to the same address with derived $\tilde{P}_{guilty}$ value higher than 0. The access is predicted to hit only if s is strictly lower than W also in the subsequence between those two accesses. Otherwise, the access is considered a miss and the computed $\tilde{P}_{guilty}$ value is distributed over the subsequence. In the example, A_1^{100} is a miss (the first access to an address is always a miss) and A_2^{100} is a hit ($s = 1 < W$). When assessing A_3^{100}, CCP-RM searches for previous accesses until reaching A_1^{100} (A_2^{100} is ignored as it is predicted not to change cache state) and evaluate that A_3^{100} may miss since $s = 2 \geq W$ due to accesses B_1^{101} and C_1^{102}.

2) The Guilt Table: Each eviction of an address in a program under a conflictive placement, which occurs with estimated probability $\tilde{P}_{guilty}$, is caused by a set of addresses. The share of responsibility of each of these addresses on causing the eviction is quantified with their $guilt$ value. While $guilt$ describes the pair-wise relation between addresses, the predicted impact of the group of addresses will depend on the relation between each pair of addresses in a group. Computed $guilt$ values are stored in the Guilt Table (GTAB), which is later queried (either by inspecting exhaustively all possible address combinations or a subset of them as proposed in Section IV-A3) to derive predicted impacts of address groups.

The GTAB is organized as a matrix. For each address $A^{seg(A)}$ in a program, the GTAB keeps track of:

- The overall likelihood of that address to miss due to conflicting with other addresses under CCP ($GTAB[A^{seg(A)}].\tilde{P}_{guilty}$). It is to the accumulated $\tilde{P}_{guilty}$ values of each access to address $A^{seg(A)}$.
- The overall guilt of each other address on potential evictions of address $A^{seg(A)}$. For instance, cell $GTAB[A^{seg(A)}].guilt[B^{seg(B)}]$ keeps $guilt$ values of address $B^{seg(B)}$ w.r.t. misses of all accesses to address $A^{seg(A)}$ during the execution of the program.

To populate GTAB, CCP-RM iterates through the sequence of memory accesses and computes their $\tilde{P}_{guilty}$ value. If for a given access to address $A_i^{seg(A)}$ $\tilde{P}_{guilty} \neq 0$, then this value is added to $GTAB[A^{seg(A)}].\tilde{P}_{guilty}$ and distributed among the different segments in-between $A_i^{seg(A)}$ and the previous access to $A^{seg(A)}$, i.e. $A_{i-1}^{seg(A)}$, as shown in Equation 2, using m_1 value computed in Equation 1. Then, the *guilt* assigned to a segment is added on top of the $GTAB[A^{seg(A)}].guilt$ of each intermediate accessed address that belongs to that segment. This is done because each address in a given segment can be placed in the same set as the analyzed address with identical probability. For example, let us consider the access A_2^{100} in the sequence $\mathcal{Q}_4 = \{A_1^{100}, B_1^{102}, C_1^{100}, D_1^{103}, B_2^{102}, E_1^{102}, A_2^{100}\}$, with $W{=}2$ and the group cardinality $K{=}3$. The number of distinct cache segments accessed after A_1^{100} is $s{=}2$ (segments 102 and 103). Those segments together with segment 100 exceed the cache associativity. We compute $\tilde{P}_{guilty}(A_2^{100}) = 1 - \left(\frac{1}{2}\right)^2 = 0.75$ and add it to $GTAB[A^{100}].\tilde{P}_{guilty}$. Next, we distribute 0.75 across $s = 2$ segments, such that $P_{guilt-seg} = \frac{0.75}{2} = 0.375$. The $guilt$ of all addresses belonging to segments 102 and 103 ($GTAB[A^{100}].guilt[B^{102}]$, $GTAB[A^{100}].guilt[D^{103}]$ and $GTAB[A^{100}].guilt[E^{102}]$) is then increased by 0.375.

$$P_{guilt-seg} = \begin{cases} \frac{\tilde{P}_{guilty}}{m_1}, & \text{if } m_1 > 0 \\ 0, & \text{otherwise} \end{cases} \tag{2}$$

A GTAB is derived for each cardinality K and is later inspected to compute the predicted impact of address combinations. This is done in two steps:

First, for each address A in the combination under analysis, we sort all the other addresses in the combination by their guilt on A, and take the value on the W^{th} position and keep it as M_A. The reason is that address A needs to conflict with at least W addresses to exceed the cache set space and such scenario cannot occur more times than the number of conflicts between address A and the least conflictive address with A among W of them. In the example in Table III, we observe high guilt values between addresses B^{102} and D^{103}, but not among the rest of addresses. This happens when those two addresses are interleaved together with other addresses that do not interleave systematically with these ones. In the table, $guilt.xxx$ stands for the $guilt$ values for other addresses omitted in the example for clarity.

And second, the predicted impact of an address combination is computed by applying the harmonic mean of all M_A values of addresses in a combination so to give lower rank to combinations with low M_A values, which reflects that A cannot have many conflicts if one of the other addresses cannot create many conflicts. Instead, CCP-RM seeks address groups in which conflicts occur due to the interaction among *all* of them. If a conflictive behavior occurs because of the interaction of a subset of these addresses, such combination is already accounted by CCP-RM for lower K values. For instance, in Table III the predicted impact of the combination [A^{100}, B^{102}, D^{103}, F^{104}] equals the *harmonic-mean*[5] of 20.0, 30.0, 28.0 and 15.0, which is 21.54.

3) Generation of Address Combinations: To derive a list of the conflictive combinations, CCP-RM builds a GTAB for each K value. Next, it generates possible combinations of K addresses and derives their impact as previously described. Ideally, CCP-RM would inspect all possible combinations (discarding the ones in which addresses from the same segment repeat). However, the number of combinations grows exponentially with the number of addresses. Therefore, CCP-RM adopts an algorithm (see Algorithm 1) to optimize this

[4]This assumption holds for TRc (the scope of this work) which deploy random replacement policy.

[5]The harmonic mean is a good estimator due to its sensitivity to lower values, giving lower ranking to combinations in which at least one address is not conflicting with others. Average mean, however, promotes combinations with only a subset of addresses conflicting.

TABLE III
RELEVANT FIELDS OF GTAB TO DERIVE PREDICTED IMPACT OF AN ADDRESS COMBINATION $[A^{100}, B^{102}, D^{103}, F^{104}]$.

	Pguilty	guilt.B	guilt.D	guilt.F	guilt.A	guilt.xxx	M_X
B^{102}	550.0	0.0	150.0	20.0	15.0	365.0	20.0
D^{103}	400.0	145.0	0.0	30.0	10.0	215.0	30.0
F^{104}	250.0	16.0	28.0	0.0	30.0	176.0	28.0
A^{100}	235.0	15.0	5.0	30.0	0.0	185.0	15.0

search by generating the subset of all possible combinations of addresses expected to be the most conflictive ones.

Rows in GTAB are sorted by their overall $\tilde{P}_{guilty}$ value. For each row, e.g. that for address $A^{seg(A)}$, the algorithm generates combinations of K addresses, containing $A^{seg(A)}$, belonging to addresses (rows) not yet inspected, i.e. with lower $\tilde{P}_{guilty}$. The search stops when $\tilde{P}_{guilty}$ of a row address is below 1% of the highest $\tilde{P}_{guilty}$ in the table (lines 6-8), since potential combinations could only consist of low impact addresses (each below the 1% threshold).

In each iteration, the algorithm considers the potential conflictive addresses with $A^{seg(A)}$ (lines 9-15). It excludes all addresses belonging to the same segment and those whose *guilt* value on the row address w.r.t. the total guilt on that address is below the defined significance threshold S_{th} (1% in our case). Next, the remaining addresses are grouped by the segment they belong to into *Segs*, to account for the fact that only one address from each group can belong to the same combination. Our search derives all the possible ways to select $K-1$ addresses from *Segs* groups (lines 17-18).

Then we need to explore conflicts against all the segments in each combination (lines 22-35). In order to explore all addresses of any given segment in *Segs*, we take into account their individual guilt on the address $addr$ with which we are generating combinations, and only consider one address representative (with the highest overall $\tilde{P}_{guilty}$ value) for those that have the same $GTAB[addr].guilt$ value, since their impact will be identical. When computing the probability of those combinations (lines 36-40), we account for the number of combinations that could be produced with addresses in that group (due to several of them having identical guilt value) to multiply the probability of having just one address. Considering a combination touching 3 segments: having 2 addresses with identical guilt value in a first segment, 3 in a second one, and 3 in the other, will lead to 18 potential combinations with exactly one address from each segment. Such probability is specifically computed in line 37. Still, multiplying probabilities without considering that combinations may overlap (the latter would diminishing the overall combined probability) leads to some pessimism. As shown later, this could only lead to requesting more runs than strictly required, thus not diminishing the confidence of the method. On the other side, addresses from the same *Segs* group, but with different guilt values, are considered individually.

Illustrative example. Next, we show the generation of combinations of size $K = 3$ in one iteration. We inspect the row containing address A^{100} and the set of potentially conflictive addresses $[E^{102}, C^{103}, B^{102}, D^{103}]$ with their corresponding values of *guilt* on A^{100}: [15, 15, 15, 20] and $\tilde{P}_{guilty}$: [72.5, 62.5, 58.75, 30.375], as illustrated in Table IV.

Addresses are grouped in *Segs* 102 and 103. The only way to make a combination of $K-1$ addresses is to select one address from each segment. *Segs* 102 contains two addresses with identical *guilt* value, thus it returns the address with

Algorithm 1 Generation of Address Combinations

```
1: Input: K        ▷ number of addresses in a combination;
          GTAB     ▷ address Guilt Table derived for K;
          S        ▷ number of cache sets;
2: Output: List of <combination; prob.> pairs;
           List of predicted impacts for each element in
           pairs listPI;

3: sort GTAB row-wise by $\tilde{P}_{guilty}$;
4: N ← nrows(GTAB);
5: for addr ← 1:N do
6:     if GTAB[addr].$\tilde{P}_{guilty}$ < 0.01*GTAB[1].$\tilde{P}_{guilty}$ then
7:         break;
8:     end if
9:     totalGuilty ← $\sum_{j=1}^{N}$GTAB[addr].guilt[j];
10:    addrSpace ← [ ];
11:    for m ← (addr+1):N do
12:        if (m ∉ getSeg(addr) and
              GTAB[addr].guilt[m] ≥ $S_{th}$*totalGuilty) then
13:            addrSpace.add(m);
14:        end if
15:    end for
16:    <combination; probability> pairs ← [ ];
17:    Segs ← list distinct segments in addrSpace;
18:    combsSeg ← list all combinations of (K-1) segments
                  from Segs;
19:    for all cs: cs ∈ combsSeg do
20:        listA ← [ ];
21:        cntA ← [ ];
22:        for seg ← cs[1]:cs[K-1] do
23:            listA[seg] ← [ ];
24:            cntA[seg] ← [ ];
25:            for all g: distinct guilt values in seg do
26:                listTmp ← [ ];
27:                for all a: a ∈ seg do
28:                    if (GTAB[addr].guilt[a] == g) then
29:                        listTmp.add(a);
30:                    end if
31:                end for
32:                listA[seg].add(address with the highest
                                  $\tilde{P}_{guilty}$ in listTmp);
33:                cntA[seg].add(number of addresses in
                                 listTmp);
34:            end for
35:        end for
36:        for all combinations < $a_1$, $a_2$,...,$a_{k-1}$, addr>:
                $a_j$ ∈ listA[j], where j=1,...,K-1 do
37:            prob ← $S * (\frac{1}{S})^K * \prod_{j=1}^{K-1}$ cntA[getSeg($a_j$)][$a_j$];
38:            pairs.add(<< $a_1$, $a_2$,...,$a_{k-1}$, addr>; prob>);
39:            listPI.add(predicted impact of
                          < $a_1$, $a_2$,...,$a_{k-1}$, addr>);
40:        end for
41:    end for
42: end for
```

TABLE IV
THE RELEVANT FIELDS OF GTAB TO GENERATE COMBINATIONS OF ADDRESSES CONTAINING ADDRESS A^{100}

		Seg 100	Seg 102		Seg 103	
	Pguilty	guilt.A	guilt.B	guilt.E	guilt.C	guilt.D
A^{100}		0.0	15.0	15.0	15.0	20.0
E^{102}	72.500					
C^{103}	62.500					
B^{102}	58.750					
D^{103}	30.375					

higher $\tilde{P}_{guilty}$, which is E^{102} and marks that two addresses share this behavior. *Segs* 103 returns addresses C^{103} and D^{103} since they have different *guilt* value. If p is the probability that three addresses are mapped to the same set, then the step will result in the next <combination; probability> pairs: [A^{100},E^{102},C^{103}; $2p$] and [A^{100},E^{102},D^{103}; $2p$]. Here $2p$ is the probability of combining the 2 addresses in segment 102 with the specific address in segment 103.

Finally, the address combination search returns the list of those T <combination; probability> pairs with the highest predicted impact in $listPI$, where the list is derived as shown in lines 36-40 of Algorithm 1 and predicted impact computed as described in Section IV-A2 (see example in Table III). While these combinations are expected to produce the highest impact in terms of number of misses when placed in the same set, their actual impact needs to be determined since *guilt* and $\tilde{P}_{guilty}$ are estimators that are used just to rank address combinations.

While we simplified Algorithm 1 for the sake of readability, its implementation can be further optimised, which we did in our experiments. We recommend the recursive implementation to generate combinations (line 18). Finding distinct segments of addresses (line 17) can be done while iterating through addresses (lines 11 and 12) by storing segments into a structure with non-repeated values (e.g. a set). *listA* and *cntA* (lines 22-34) can be created in line 13, where the user does not need to keep all the addresses with the same *guilt*, but only K with the highest $\tilde{P}_{guilty}$ value. For efficient searching of relevant data we recommend the use of map structures (instead of lists/arrays).

B. Impact and Probability Calculation

The algorithm from the previous step derives a list of <combination, probability> pairs, describing representative address combinations and the probability of observing them or any other with the same predicted impact. The impact in terms of miss count of each representative address combination is evaluated with a cache simulator. The addresses in the combination are mapped to the same set, while others are randomly mapped. Several Monte-Carlo simulations are performed and the impact of the given combination is determined as the average impact across the different Monte-Carlo simulations.

Next we map <combination, probability> pairs into the <probability, misscount> domain by ordering the pairs per derived impact, and computing the combined probability and impact for the first pair, first two pairs, first three pairs, and so on, until we cover all pairs in the list. Given a number of first N address combinations, the combined miss count (having one of them) is their average miss count, and the combined probability is the union of their probabilities. Since their individual probabilities do not have to be necessarily disjoint, to determine the exact joint probability one would need to determine the overlaps between all groups of 2, 3,...,N combinations. However, to avoid the inherent computational complexity of such activity, similar to the previous step, we upperbound such probability as the addition of their probabilities, which is generally a tight upperbound. The individual probabilities of all combinations of K addresses are identical, so determining the joint probability becomes trivial.

C. Assessment Against the pWCMC Curve

Previous steps result in a pair <probability, misscount> for each CCP, i.e. combination and group of combinations deemed as conflictive. As next step, we generate a probabilistic worst-case miss-count (pWCMC) by applying Extreme Value Theory to the miss counts in a sample of R randomly generated RM mappings. We check whether the pWCMC distribution upperbounds all CCP (i.e their miss count). If this is the case, the default number of measurements R used by MBPTA suffices to derive trustworthy WCET estimates. Otherwise, more runs are performed, until pWCMC upperbounds all pairs. The obtained number of runs R' is finally returned as the number of runs to be collected by the end user.

For instance, in Figure 5, the solid curve is the pWCMC estimate and the black squares and crosses the miss counts obtained for all CCP whose probability of occurrence is above P_{rel}. Black circles are those CCP below P_{rel} and hence, are not considered. Black squares are those CCP whose miss count is covered by the pWCMC, while the miss counts of the CCP marked with crosses are not. In this example scenario, *CCP-RM* requires the user to increase the number of runs from R to R' such that the impact of those combinations (black crosses) is properly upperbounded. We also observe that when increasing the number of runs to R' runs (with $R' > R$) the resulting pWCMC curve captures the impact of all relevant combinations (black squares and crosses). Overall, this results in an increased number of runs R' for which the obtained pWCMC estimate reliably upperbounds the miss count of all combinations and therefore, the pWCET estimate obtained with R' runs also upperbounds their timing impact.

D. Miss Count - Execution Time correlation

It could be the case that the pWCMC curve upperbounds CCP systematically with the default number of runs R. This can occur either because R is sufficient to capture all CCP, or simply because the impact of CCP is lower than that of other *jres*. In the latter case, as shown in [2], other *jres* are expected to occur with a sufficiently high probability to be captured with the default number of runs of MBPTA. However, in general, miss counts have a relevant impact (if not the highest impact) on execution time amongst the existing *jres*, and so techniques

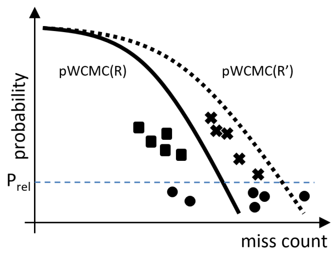


Fig. 5. Illustrative application of *CCP-RM* and *CCP-hRP*.

TABLE V
PEARSON AND SPEARMAN COEFFICIENTS FOR $NormMiss$ AND $NormET$.

	CCP-hRP		CCP-RM	
	Pearson	**Spearman**	**Pearson**	**Spearman**
`a2time`	0.997	0.992	0.976	0.970
`aifftr`	0.918	0.911	0.969	0.960
`aifirf`	0.960	0.956	0.988	0.986
`aiifft`	0.923	0.913	0.960	0.952
`basefp`	0.999	0.998	0.952	0.933
`bitmnp`	0.998	0.998	0.975	0.960
`cacheb`	1.000	0.970	0.992	0.988
`idctrn`	0.950	0.951	0.969	0.973
`iirflt`	0.997	0.979	0.977	0.997

like CCP-RM are needed. Whether execution time correlates with miss counts has already been proven in many platforms. In our particular case, we have quantitatively assessed the correlation between miss counts and execution time in our FPGA platform, whose specific setup is shown in Section VII. To that end we used Pearson product-moment correlation coefficient and Spearman's rank correlation coefficient. The former measures the linear dependence between two variables, while the latter measures the statistical dependence between two variables by assessing to what extent those variables can be modeled using a monotonic function. The outcome of both methods is a value $o \in [-1, 1]$. For $o = 1$ there is total positive correlation; for $o = 0$ no correlation is found; and for $o = -1$ total negative correlation i reported. In our experiments we use a 5% significance level (a typical value for this type of tests [19]). We used normalised values for qualitative assessment, where we plotted normalised values of execution times/misses and compared the trends by visual inspection. Table V shows that all EEMBC benchmarks result in high values of correlation coefficients by both methods confirming that miss counts and execution times are highly linearly correlated, for both RM and hRP (presented in next Section). The lower o values occur for benchmarks with low miss count variations. Thus, other sources of jitter, like those introduced by the store buffer, have a relatively higher impact than for other benchmarks.

V. THE CCP-HRP MECHANISM

CCP-hRP is analogous to CCP-RM but with some critical differences due to the different behaviour of hRP and RM.

A. Deriving Relevant Address Combinations

As in the case of CCP-RM, CCP-hRP produces, for each $K > W$ address count, a list of address combinations that, if placed in the same set, produce high miss counts. Note that, similar to CCP-RM, only T combinations are kept. To build those lists, CCP-hRM builds upon the address Guilt Table (GTAB), but $\tilde{P}_{guilty}$ and so $guilt$ are computed differently to the case of RM caches.

First, $\tilde{P}_{guilty}$ is obtained as described in Equation 3. While the formulation is analogous to that of CCP-RM (see Equation 1), there is a key difference. In the case of RM caches, m_1 is set based on the number of different *segments* (s) accessed in between two consecutive accesses to a given cache line. In the case of hRP caches, m_2 is set based on the number of different *cache lines* (q) accessed in between.

$$\tilde{P}_{guilty} = 1 - \left(\frac{W-1}{W}\right)^{m_2} \quad m_2 = \begin{cases} 0, & \text{if } q < W \\ q, & \text{if } W \leq q < K \\ K-1, & \text{otherwise} \end{cases} \tag{3}$$

For instance, in the example for RM, in the sequence $A_1^{100}, B_1^{102}, C_1^{102}, A_2^{100}$ we would have $s = 1$, but $q = 2$. This reflects the fact that for hRP caches, A^{100}, B^{102} and C^{102} can compete for the space in the same set, whereas for RM caches B^{102} and C^{102} can never be placed in the same set.

1) Guilt estimation: When for an access A_i $\tilde{P}_{guilty} \neq 0$, its value is 'distributed' among the intermediate accesses between $A_i^{seg(A)}$ and $A_{i-1}^{seg(A)}$. Each accessed address is assigned a guilt value w.r.t. address $A^{seg(A)}$ computed as shown in Equation 4, using $\tilde{P}_{guilty}$ and m_2 as described in Equation 3. Note that in this case $guilt$ considers cache line addresses individually rather than cache segments, as it is the case in Equation 2.

$$guilt = \begin{cases} \frac{\tilde{P}_{guilty}}{m_2}, & \text{if } m_2 > 0 \\ 0, & \text{otherwise} \end{cases} \tag{4}$$

For example, let us recall the previous example for CCP-RM and consider the access A_2^{100} in the sequence $\{A_1^{100}, B_1^{102}, C_1^{100}, D_1^{103}, B_2^{102}, E_1^{102}, A_2^{100}\}$, $W = 2$ and the group cardinality $K = 3$. The number of distinct addresses accessed after A_1^{100} is $q = 4$. Those addresses together with A^{100} exceed the cache associativity. Since $q \geq K$, we compute $\tilde{P}_{guilty}(A_2) = 1 - \left(\frac{1}{2}\right)^2 = 0.75$ using $m_2 = K - 1 = 2$. We distribute this to $q = 4$ addresses, such that $guilt = \frac{0.75}{2} = 0.375$. $Guilt$ of all addresses in between (B^{102}, C_1^{100}, D^{103} and E^{102}) is then increased by 0.375. Note that the addition of guilt assigned to intermediate accesses is bigger than $\tilde{P}_{guilty}$. The idea is that for $K = 3$, CCP-hRP (and CCP-RM) constructs 3-address combinations that in this case can be any of ABC, ABD, ACD, BCD, ABE, ACE, etc. 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 CCP-hRP, 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 CCP-hRP iterative process – those other intermediate accesses will be considered simultaneously.

2) The Guilt Table: In the case of CCP-hRM, the GTAB has the same characteristics as for CCP-RM: as many rows and columns as different (cache line) addresses are accessed in the program to store guilt values and one additional column to track $\tilde{P}_{guilty}$ values. Cell $GTAB[A].guilt[B]$ 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 $GTAB$ is built for every value of K. From the $GTAB$ we infer information about the impact that each address has on the evictions of each other address. However, that the actual values in each cell of GTAB differ between CCP-hRP and CCP-RM, since they operate on different formulations for $\tilde{P}_{guilty}$ and $guilt$.

3) Generation of Address Combinations: The approach to generate address combinations from GTAB is very similar to that of CCP-RM, but with some key differences. For instance, when considering conflictive addresses with the address of the corresponding row, we only consider those addresses 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} is 1% as for CCP-RM. Note that whether addresses belong to the same or different segments is irrelevant for CCP-hRP, since addresses can be randomly placed in the same set regardless of their segment. In Algorithm 1, the condition from line 12 becomes redundant.

We group addresses into *buckets* in each row of the GTAB, grouping all the addresses with the same guilt value w.r.t. the address of that row. The objective, as explained next, is limiting the number of address combinations that need to be considered. In case this originates too many buckets, thus ending up requiring high computation time, then the grouping criterion can be tweaked to tolerate small differences between guilt values of addresses in the same bucket, as a means to reduce their total count. In Algorithm 1, buckets correspond to the listTmp, with a different bucket created for each distinct guilt value in addrSpace (not per segment), and listTmp are global variables.

Then, 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 (and modify line 36 in the algorithm accordingly). 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 $b1$ plus 1 from $b2$, 1 from $b1$ plus 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 of 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 deserve to be explored by simulation (line 37 of Algorithm 1). The probability equals to the probability of an individual address combination, $S * (1/S)^K$, times the number of different combinations that can be produced with the addresses of a bucket.

B. Impact and Probability Calculation

When all addresses have been analyzed and the list with $T = 20$ combinations[6] for a particular value of K is obtained, we perform cache simulations to determine their miss counts. 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.

The approach to determine the probability of each combination is analogous to that of CCP-RM: $S \times (1/S)^K$ for a single combination. For the combined probability (when the number of combinations represented by the one in the list is more than one), we pessimistically use the addition of their individual probabilities as for CCP-RM.

C. Assessment Against the pWCMC Curve

As for CCP-RM, CCP-hRP 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. The number of simulations, R, is determined by MBPTA.

VI. Evidence for Certification

The mechanisms we propose to identify and capture CCPs are meant to enhance the reliability of MBPTA results, with a view to meeting the reliability requirements of the V&V of embedded critical systems. Not surprisingly, both techniques feature a heuristic search over the CCP space: a reasoned heuristic-based empirical evidence is *de-facto* the only means to analyze overly-complex hardware and software systems, where providing exhaustive evidence is generally untenable. This section aims at showing that CCP-RM/hRP can concretely improve the reliability of MBPTA. We do so by conducting a three-fold assessment.

1) We show that the implemented heuristics are accurate by comparing their outcome with that of ReVS [32], which is known to provide *exact* results by exhaustively exploring the impact of all address combinations with cardinalities higher than cache associativity. Since the cost of ReVS is prohibitive for real-size programs (hence the need for a heuristic), we stick to a controlled scenario as explained next.

[6]One combination may be the representative of 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.

2) We show the effectiveness of CCP-RM/hRP in detecting otherwise ignored CCPs, and the impact this has on the overall number of runs CCP-RM/hRP require to use (R') as compared to the default number of runs used by MBPTA (R).
3) We show that the benefits of CCP-RM/hRP come with affordable computational requirements.

Note that, in our case, the implication of using a heuristic is that the CCP impact computed by CCP-RM/hRP may slightly differ from that computed by the exact method. However, our algorithms do not focus on a single CPP but on a list thereof, and the list of CCPs considered by the heuristics is always including the topmost (highest-impact) CCPs. For the benchmarks evaluated in this section and case study (next section) we consider the topmost 20 CCPs (i.e., $T=20$), which in practice results in considering already more combinations strictly required, as confirmed by the results. We inspect combinations of cardinalities K in the range $W+1 \leq K \leq 13$ (probabilities of address combinations of higher cardinalities are negligible).

A. Experimental Setup

We use EEMBC Autobench benchmark suite [37], representative of some real-time automotive applications. On average EEMBC Automotive benchmarks comprise 6,500 Lines of Code, where the average number of distinct addresses is 2,500 for instructions and 5,600 for data. We use a simulation environment based on the cycle-accurate SoCLib [41] framework. We model an architecture featuring a pipelined in-order processor with separated instruction (IL1) and write-back data (DL1) caches, both deploying random replacement, and RM or hRP policies. Access latencies for IL1 and DL1 are 1 cycle for hits and 4 cycles for misses, which sums up to memory latency, for a total of 20 cycles.

B. CCP-RM/hRP Accuracy

ReVS [32] *precisely* identifies all conflictive cache placements, by exhaustively exploring (measuring) the impact of all possible address combinations rather than predicting their impact, as done by CCP-RM/hRP. Thus, ReVS is used as a reference to assess the accuracy of CCP-RM/hRP in identifying CCPs based on heuristics. As different address combinations can cause similar miss impact, we are not comparing individual address combinations and, instead, we compare the computed <probability, misscount> pairs. The comparison is conducted both (i) visually, by plotting pairs as in Figure 6 (we restrict to the `cacheb` benchmark for space constraints), and (ii) analytically, considering the size of the sample required to upper bound those pairs (shown in Tables VI and VII). To fairly compare the sample size needed to upper bound pairs and discard the variation due to the convergence criteria, we analyze the same measurement samples with both CCP-RM/hRP and ReVS methods. The cost of ReVS is however prohibitive for real-size programs, so for the sake of comparison only, we focus on a controlled scenario with a small number of addresses, which was obtained by creating synthetic benchmarks accessing the 15 most-accessed cache lines from each EEMBC Automotive benchmark.

This limitation on the number of addresses makes that the number of accessed pages reduces to 1 or 2 cache segments for most benchmarks, making that no CCP exist for RM in most of the cases. Hence, only for this controlled scenario and RM, we use a small cache (512B 32B/line 2-ways IL1/DL1), such that even programs with small address footprints can exhibit a conflictive behaviour. Note that with hRP 15 addresses can easily exceed cache associativity. In the case of hRP reducing cache size would further increase the probability of CCP, which would easily make the default number of runs of MBPTA sufficient, so no insight would be shown if cache size was decreased, as done for RM. Therefore in the controlled scenario we use a 4KB 32B/line 2-way IL1 and DL1 cache setup.

TABLE VI
R' FOR CCP-RM AND REVS IN CONTROLLED SCENARIO

	R'_{IL1}		R'_{DL1}		R'	
	ReVS	**CCP-RM**	**ReVS**	**CCP-RM**	**ReVS**	**CCP-RM**
`a2time`	1,460	1,460	8,650	8,650	8,650	8,650
`aifftr`	480	480	670	670	670	670
`aifirf`	6,300	6,300	300	300	6,300	6,300
`aiifft`	410	410	8,500	8,500	8,500	8,500
`basefp`	420	420	300	300	420	420
`bitmnp`	370	370	3,570	3,570	3,570	3,570
`cacheb`	300	300	690	690	690	690
`idctrn`	300	300	300	300	300	300
`iirflt`	300	300	300	300	300	300

Table VI reports the minimum number of measurements (R') deemed sufficient for a reliable application of MBPTA according to CCP-RM and ReVS for IL1 and DL1. The final number of runs required for each technique is the maximum of both (IL1 and DL1). The accuracy of CCP-RM is confirmed by the fact that *in all cases it computes the same values as* ReVS. In general, this is expected since CCP-RM successfully identifies the critical combinations, whereas ReVS necessarily identifies them due to its brute force nature. Hence, by applying the same requirements on the pWCMC with both methods and using the same measurement sample, MBPTA converges with the same number of runs. In particular, when CCP-RM identifies that conflictive placement can occur, it returns the same address combinations that are top-ranked by ReVS, or with a very close miss-impact. For the IL1, for six benchmarks (`ia2time`, `iaifftr`, `iaifirf`, `ibitmnp`, `icanrdr`, `iidctrn`) CCP-RM does not return any conflictive address combination: this is explained by the fact that in those cases all address combinations returned by ReVS are already upper-bounded with the default number of measurements by MBPTA. For other benchmarks CCP-RM also returns no address combinations as potentially conflictive for high cardinalities of K, while ReVS does. This may happen when the conflictive impact is actually caused by address groups smaller than K that are instead considered by CCP-RM.

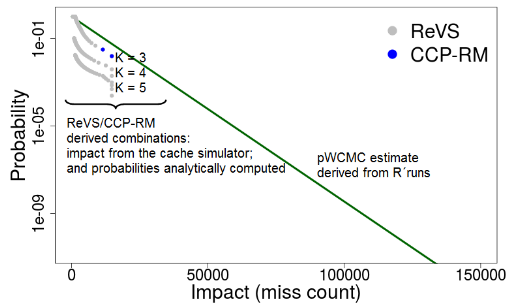


Fig. 6. pWCMC for `cacheb` and a DL1 RM cache.

For instance, Figure 6 shows that, although conflictive placements – referred to as address combinations – can occur with cardinalities $K = 4$ and $K = 5$, as recognized by ReVS, those placements are upper bounded by the truly conflictive placement of combinations with cardinality $K = 3$.

Table VII shows the number of runs that each of the methods regards as the minimum to use for a reliable MBPTA application according to CCP-hRM and ReVS. 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, CCP-hRM 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'.

TABLE VII
R' FOR CCP-HRP AND REVS IN CONTROLLED SCENARIO

	R'_{IL1}		R'_{DL1}		R'	
	ReVS	**CCP-hRP**	**ReVS**	**CCP-hRP**	**ReVS**	**CCP-hRP**
`a2time`	58,360	58,360	540	540	58,360	58,360
`aifftr`	6,840	6,840	5,500	5,500	6,840	6,840
`aifirf`	21,390	21,390	11,530	11,530	21,390	21,390
`aiifft`	8,920	8,920	8,770	8,770	8,920	8,920
`basefp`	82,080	82,080	20,010	20,010	82,080	82,080
`bitmnp`	4,640	4,640	3,510	3,510	4,640	4,640
`cacheb`	18,610	18,610	7,950	7,950	18,610	18,610
`idctrn`	65,770	65,770	47,700	47,700	65,770	65,770
`iirflt`	18,310	18,310	49,760	49,760	49,760	49,760

Despite using different setups, CCP-RM requires fewer runs than CCP-hRP. This occurs because CCP for RM (if any) can only occur with few addresses because there are few cache segments. Therefore, the probability to capture those CCP is relatively high and few runs suffice in general. Conversely, with hRP it is often the case that CCP involve more addresses than those for RM. Therefore, their probability of occurrence is lower and thus, a larger number of runs is required to guarantee that they are effectively captured.

C. Evaluation of CCP-RM and CCP-hRP Effectiveness

With the purpose of entailing an increase in the number of conflicts and hence, further stressing CCP-RM/hRP, in this section we evaluate CCP-RM/hRP for a relatively small cache 4KB 32B/line 2-ways IL1 and DL1 cache. We also focus on full-size EEMBC benchmarks to assess whether our methods effectively improve MBPTA reliability by identifying potentially unobserved CCP. To that end, we compare the number of measurements required by a default application of MBPTA (R) against those required by CCP-RM/hRP (R'), for a set of EEMBC benchmarks. Note that for this experiment using all addresses, ReVS could not be used due to its exponential execution time requirements with the number of benchmarks' number of unique addresses.

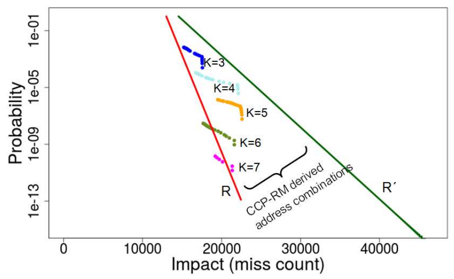
TABLE VIII
CCP-RM ON EEMBC ('LHOOD' STANDS FOR LIKELIHOOD)

	CCP-RM				**MBPTA**	
	R'_{IL1}	R'_{DL1}	R'	lhood(R')	R	lhood(R)
`a2time`	300	730	730	10^{-9}	300	$2.00 * 10^{-4}$
`aifftr`	300	6,160	6,160	10^{-9}	300	$3.64 * 10^{-1}$
`aifirf`	470	22,490	22,490	10^{-9}	370	$7.11 * 10^{-1}$
`aiifft`	300	110,000	110,000	10^{-9}	74,600	$7.88 * 10^{-7}$
`basefp`	320	1,120	1,120	10^{-9}	960	$1.93 * 10^{-8}$
`bitmnp`	300	310	310	10^{-9}	300	$1.95 * 10^{-9}$
`cacheb`	460	390	460	10^{-9}	500	$< 10^{-9}$
`idctrn`	350	1,050	1,050	10^{-9}	500	$5.18 * 10^{-5}$
`iirflt`	300	930	930	10^{-9}	320	$8.00 * 10^{-4}$

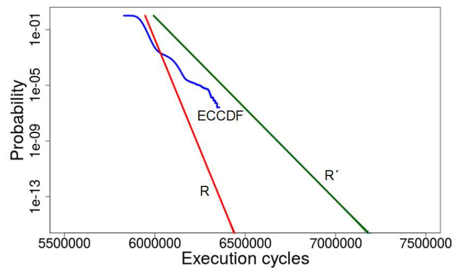
For CCP-RM Table VIII reports different values of R and R'. Only for one benchmark (`cacheb`) the default MBPTA application asks for more runs than those actually needed to observe conflictive placements. In this case, 460 runs suffice for MBPTA to converge in the cache miss domain, but few more runs are needed in the execution time domain to have enough high execution time values to converge due to variations across random samples. For the remaining eight benchmarks, collecting R measurements results in a probability of not capturing CCP for RM higher than the established threshold, see $lhood(R)$ column. In fact, for two benchmarks (`aifftr`, `iirflt`) we observe that the EVT projection using R does not upperbound the ECCDF applied on an arbitrarily large number of runs, which is actually upperbounded when the pWCET is estimated using R' runs as determined by CCP-RM. Figure 7 reports the results for one of those benchmarks, `aifftr`.

In Figure 7(a) the pWCMC estimate derived with R runs and R' is plotted against the conflictive sets of addresses (from 3 to 7 addresses) found by CCP-RM for DL1. As it can be observed, for the default R the pWCMC does not cover all relevant address combinations, while with the CCP-RM-provided R' the resulting pWCMC does upperbound all conflictive address combinations.

In the time domain we take as term of comparison the Empirical Complementary CDF (ECCDF) derived from four



(a) pWCMC estimate with R and R' runs



(b) pWCET estimate with R and R' runs

Fig. 7. EVT projections for benchmark `aifftr` with RM caches

million execution times we collected. Figure 7(b) shows that the pWCET curve obtained from R runs does not upperbound the ECCDF. The pWCET obtained with R' runs returned by CCP-RM, instead, upperbounds the ECCDF.

Analogous results are reported in Table IX for CCP-hRP. 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, but indicates that there is non-negligible risk of not observing some high-impact timing events in the analysis runs if CCP-hRP is not used.

When comparing the number of runs of CCP-hRP 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`) 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, CCP-hRP can deal with arbitrary access patterns without any explicit limit. Thus, CCP-hRP removes the uncertainty brought by ReVS due to non-analyzed addresses.

D. *CCP-RM and CCP-hRP Time Requirements*

The main execution time requirement of CCP-RM, CCP-hRP, and ReVS comes from the cache simulations (Section IV-B). In order to run the simulations, we used a cluster running 100 jobs in parallel. For the controlled scenario, CCP-RM and CCP-hRP require 1 and 11 minutes respectively per program on average out of which 14 and 4 seconds are spent in deriving conflictive placements respectively. ReVS, due to its complete exploration approach, required 2 and 27 hours for RM and hRP cache setups respectively. With full benchmarks, CCP-RM and CCP-hRP require on-average 18 and 38 minutes per benchmark respectively out of which 2.5 and 1 min on average are spent to calculate conflictive placements respectively. ReVS simulations would take years to be executed. Our results show that CCP-RM and CCP-hRP result in affordable execution time requirements.

TABLE IX
CCP-hRP ON EEMBC

	CCP-hRM				MBPTA	
	R'_{IL1}	R'_{DL1}	R'	lhood(R')	R	lhood(R)
`a2time`	67,150	300	67,150	10^{-9}	300	$9.11 * 10^{-1}$
`aifftr`	300	4,760	4,760	10^{-9}	300	$2.71 * 10^{-1}$
`aifirf`	20,080	8,090	20,080	10^{-9}	14,260	$4.06 * 10^{-7}$
`aiifft`	300	10,630	10,630	10^{-9}	300	$5.57 * 10^{-1}$
`basefp`	78,220	300	78,220	10^{-9}	1,250	$7.18 * 10^{-1}$
`bitmnp`	330	1,800	1,800	10^{-9}	300	$3.16 * 10^{-2}$
`cacheb`	19,840	1,500	19,840	10^{-9}	9,360	$5.69 * 10^{-5}$
`idctrn`	67,460	43,040	67,460	10^{-9}	300	$9.12 * 10^{-1}$
`iirflt`	29,920	2,430	29,920	10^{-9}	300	$8.12 * 10^{-1}$

VII. Railway Case Study on FPGA

We also evaluate CCP-RM/hRP with a real industrial case study from the railway domain, running on a LEON3 FPGA board modified to support RM and hRP caches.

Instruction and data addresses are collected with the standard debug interface (DSU) support present in our FPGA board. Notably, other processor architectures provide similar or more advanced tracing features, e.g. the Nexus Interface for NXP processors and Coresight for ARM processors. Since address tracing can affect execution time, time traces are collected only with the address tracing mechanism disabled. Address traces are collected in a single separate run, for which timing is not considered. Obtained data and instruction traces can be used to perform CCP-RM/hRP analysis and random-cache simulations since cache-set mapping is now independent from the address.

The railway application implements a safety function from the European Train Control System (ETCS) reference architecture. The analysed application controls the execution of all safety functions associated to the travelling speed and distance supervision. This safety function is provided with the highest integrity level defined in the railway safety standards, SIL-4, and has strict real-time requirements. The end user provided us input vectors that exercise 10 different paths (`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.

A. *Experimental Results*

The case study was executed by the end user on the FPGA (following the exact specifications of the FPGA setup). Both first level caches are 16KB 4-way, with 32B cache line size for instructions and 16B for data.

TABLE X
CCP-RM AND CCP-HRM RESULTS ON THE RAILWAY CASE STUDY.

	CCP-RM			MBPTA	**CCP-hRP**			MBPTA
	R'_{IL1}	R'_{DL1}	R'	R	R'_{IL1}	R'_{DL1}	R'	R
TEST0	1,560	300	1,560	300	300	1,300	1,300	370
TEST1	300	410	410	300	600	3,800	3,800	3,800
TEST2	300	340	340	300	600	1,000	1,000	300
TEST3	350	300	350	300	1,600	850	1,600	300
TEST4	—	—	300	300	1,200	1,100	1,200	750
TEST5	10,300	300	10,300	300	2,100	900	2,100	480
TEST6	—	3,200	3,200	300	500	890	890	890
TEST7	1,240	300	1,240	300	500	4,400	4,400	300
TEST8	—	—	300	300	700	2,300	2,300	300
TEST9	—	300	300	300	4,800	1,740	4,800	1,740

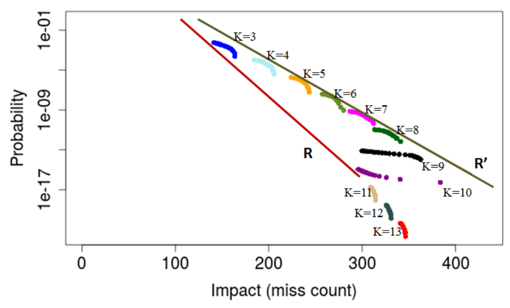


Fig. 8. pWCMC for *TEST7* (DL1) by applying MBPTA (R) and CCP-hRP+MBPTA (R').

CCP-RM. Table X (left) reports the number of runs identified by CCP-RM (R') for IL1, DL1 and globally, against the default number required by MBPTA for convergence (R). Both R and R' measurements suffice to upper bound all CCP in the RM setup. Since the number of segments for each benchmark is generally low (between 4 and 13), conflictive behaviour can occur only for combinations of addresses of low cardinality, which are more likely to be observed already with a moderate number of runs. For data traces of two benchmarks (TEST8 and TEST9), the number of segments does not exceed cache associativity. For data trace of TEST2 and instruction traces of groups of benchmarks (TEST0, TEST1, TEST2, TEST3, TEST5, TEST6, TEST7 and TEST9) the method does not identify any CCP: despite CCP can be theoretically had, as the number of segments exceeds associativity, the addresses from distinct segments are barely interleaved, hence CCP-RM attaches small guilt values. For data traces of (TEST0, TEST1, TEST3, TEST5, TEST7), the method identifies relevant combinations at most for a single K value (5 in all cases apart from TEST5 with 6). However, each of them are already upper bounded by the number of runs required by a default MBPTA application, 10,300 at most. Further, by comparing the EVT projection with the actual impact (ECCDF projection over 10 million observed miss counts), we observed that pWCMC estimates are very close to actual values.

CCP-hRP. Table X (right) reports the results we obtained, in terms of the number of runs that MBPTA and CCP-hRP require in the miss domain for the hRP setup. As it can be seen, the default application of MBPTA failed to upper-bound some address combinations for data or 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 shown in Figure 8 for TEST7 and the DL1 where CCP-hRP <probability, misscount> pairs (points in the plot) are not upper-bounded by the pWCMC curve (lower straight line in the plot) when using $R = 300$, as determined by MBPTA. Instead, if we use $R' = 4,400$, as determined by CCP-hRP, the pWCMC curve properly upper-bounds those pairs.

B. *CCP-RM/hRP Execution Time Requirements*

CCP-RM took less than 0.5s per input vector to identify conflictive placements and less than 5s on average for cache simulations. CCP-hRP, instead required 1.3 minutes per input vector to derive the conflictive combinations and 20 seconds per input vector for cache simulations. As expected, the cost on the hRP setup is a bit higher than on the RM setup due to the increased number of CCPs in the former since addresses in the same segment may potentially collide.

VIII. RELATED WORK

While several approaches exist for attacking the WCET estimation problem [44], we focus here on related work on MBPTA. MBPTA deals with *jres* by either enforcing some hardware components to always operate in their worst-case mode (upperbounding) or by injecting time randomization in those components exhibiting larger variability. The realization of the above concepts led to the notion of MBPTA-compliant hardware, as formalized in [15], which has been later implemented either by low-overhead modifications in the hardware design of resources like caches [27] or buses [15], or by resorting to lightweight software means to obtain the same effects of randomization, as it has been done in the case of deterministic caches [26].

A time-randomized cache, deploying hRP placement, was firstly introduced in [27]. Its improved version, RM, has been proposed in [22]. Previous work observed that, in certain scenarios, TRc can lower the confidence had on WCET estimates [2], [31], [39]. A detection mechanism for such scenarios and ways to account for them in the analysis have been proposed for hRP cache designs and programs in which accesses to memory are homogeneously distributed [2], and for programs with arbitrary access patterns but extremely limited size [32].

Several studies focus on properly handling control-flow and data dependencies [28], [45] and controlling execution time dependence on input-data [13]. Other works considered the application of EVT or other probabilistic techniques to programs running on deterministic (non-randomized) architectures [4], [10], [13], [20]. In that scenario, however, the lack of randomization (and upperbounding) necessarily leads to extremely fragile results in terms of representativeness of the different *jres*, including caches.

Statistical aspects of MBPTA are addressed in [40], focusing attention on EVT parameter selection and [3], [13], [35], the tail distribution to better describe WCET estimates. Other works apply MBPTA in multicore systems where shared hardware resources are MBPTA-compliant [43] or contention impact is upperbounded [9]. The open challenges for MBPTA application are explored in [25].

IX. Conclusions

We propose CCP-RM and CCP-hRP, two conflictive-placement detection mechanisms for high-performance TRc deploying RM and hRP placement respectively. CCP-RM/hRP identify the cache conflictive placements that result in high execution times. We exploit this information to derive the minimum number of measurements R' to be performed so that the probability of missing the impact of those cache conflictive placements is below a configurable threshold (e.g. 10^{-9}). The adoption of CCP-RM/hRP guarantees a cache-conflictive placement aware, reliable application of MBPTA. Our results using benchmarks and a real case study, respectively run on a simulator and a real board deploying RM and hRP caches, show the effectiveness of CCP-RM/hRP in identifying cache conflict placements and deriving an appropriate value for R'. Moreover, we show that the cost of applying CCP-RM is lower than that of CCP-hRP due to the reduced number of CCP that can occur in RM caches w.r.t. those in hRP ones.

Acknowledgments

This work has received funding from the Spanish Ministry of Science and Innovation under grant TIN2015-65316-P and the HiPEAC Network of Excellence. The Ministry of Economy and Competitiveness partially supported Suzana Milutinovic under FPI grant (BES-2016-077561), Jaume Abella under Ramon y Cajal postdoctoral fellowship (RYC-2013-14717) and Enrico Mezzetti under Juan de la Cierva-Incorporación postdoctoral fellowship (IJCI-2016-27396).

References

[1] ARM Expects Vehicle Compute Performance to Increase 100x in Next Decade. Technical report, https://www.arm.com/about/newsroom/arm-expects-vehicle-compute-performance-to-\increase-100x-in-next-decade.php, 2015.
[2] J. Abella et al. Heart of Gold: Making the improbable happen to extend coverage in probabilistic timing analysis. In *ECRTS*, 2014.
[3] J. Abella et al. Measurement-based worst-case execution time estimation using the coefficient of variation. *ACM TODAES.*, 2017.
[4] G. Bernat and M. Newby. Probabilistic WCET analysis, an approach using copulas. *Journal of Embedded Computing*, 2006.
[5] Certification Authorities Software Team. Multi-core Processors - Position Paper. Technical report, CAST-32A, November 2016.
[6] Cobham Gaisler. LEON3 Processor (Probabilistic platform). http://www.gaisler.com/index.php/products/processors/leon3.
[7] F. Corno et al. Gatto: a genetic algorithm for automatic test pattern generation for large synchronous sequential circuits. *IEEE Trans. on CAD of Integrated Circuits and Systems*, 1996.
[8] L. Cucu-Grosjean et al. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.
[9] E. Diaz et al. MC2: Multicore and cache analysis via deterministic and probabilistic jitter bounding. In *Ada Europe*, 2017.
[10] S. Edgar and A. Burns. Statistical analysis of WCET for scheduling. In *the 22nd IEEE Real-Time Systems Symposium (RTSS01)*, 2001.
[11] A. Blin et al. Maximizing Parallelism without Exploding Deadlines in a Mixed Criticality Embedded System. In *ECRTS*, 2016.
[12] E. Mezzetti et al. A rapid cache-aware procedure positioning optimization to favor incremental development. In *RTAS*, 2013.
[13] George Lima et al. Extreme value theory for estimating task execution time bounds: A careful look. In *28th ECRTS*, 2016.
[14] J. Nowotsch et al. Multi-core interference-sensitive WCET analysis leveraging runtime resource capacity enforcement. In *ECRTS*, 2014.
[15] L. Kosmidis et al. Fitting processor architectures for measurement-based probabilistic timing analysis. *Microprocessors and Microsystems*, 47:287 – 302, 2016.
[16] Michael H. Schulz et al. Socrates: a highly efficient automatic test pattern generation system. *IEEE Transactions on CAD of Integrated Circuits and Systems*, 7(1):126–137, 1988.
[17] H. Falk and H. Kotthaus. Wcet-driven cache-aware code positioning. In *Proc. of CASES*, 2011.
[18] G. Fernandez et al. Computing safe contention bounds for multicore resources with round-robin and FIFO arbitration. *IEEE Trans. Computers*, 66(4), 2017.
[19] R.A. Fisher. The arrangement of field experiments. *Journal of the Ministry of Agriculture of Great Britain*, pages 503 – 513, 1926.
[20] J. Hansen et al. Statistical-based WCET estimation and validation. In *WCET Analysis workshop*, 2009.
[21] A. et al. Hashemi. Efficient procedure mapping using cache line coloring. *ACM SIGPLAN Notices*, 1997.
[22] C. Hernandez et al. Random modulo: a new processor cache design for real-time critical systems. In *DAC*, 2016.
[23] E. Heymann. The digital car. more revenue, more competition, more cooperation. Technical report, Deutsche Bank Research, Frankfurt am Main Germany, July 2017.
[24] International Organization for Standardization. *ISO/DIS 26262. Road Vehicles – Functional Safety*, 2009.
[25] S. Jimenez et al. Open challenges for probabilistic measurement-based worst-case execution time. *IEEE ESL*, 2017.
[26] L. Kosmidis et al. Probabilistic timing analysis on conventional cache designs. In *DATE*, 2013.
[27] L. Kosmidis et al. Efficient cache designs for probabilistically analysable real-time systems. *IEEE Trans. Computers*, 2014.
[28] L. Kosmidis et al. PUB: Path upper-bounding for measurement-based probabilistic timing analysis. In *ECRTS*, 2014.
[29] S. Kotz et al. *Extreme value distributions: theory and applications*. World Scientific, 2000.
[30] P. Lokuciejewski, H. Falk, and P. Marwedel. WCET-driven Cache-based Procedure Positioning Optimizations. In *Proc. of the 20th Euromicro Conference on Real-Time Systems (ECRTS)*, 2008.
[31] E. Mezzetti et al. Randomized caches can be pretty useful to hard real-time systems. *LITES*, 2(1), 2015.
[32] S. Milutinovic et al. Modelling probabilistic cache representativeness in the presence of arbitrary access patterns. In *ISORC*, 2016.
[33] S. Milutinovic et al. Software time reliability in the presence of cache memories. In *Ada-Europe*, 2017.
[34] T. Mitra, J. Teich, and L. Thiele. Time-critical systems design: A survey. *IEEE Design &‘ Test*, 35(2), 2018.
[35] K. Palma et al. On using gev or gumbel models when applying evt for probabilistic wcet estimation. In *RTSS*, 2017.
[36] K. Pettis and R. C. Hansen. Profile guided code positioning. *ACM SIGPLAN Notices*, 25(6):16–27, 1990.
[37] J. Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[38] S. Quinton. Industrial panel: Multicore architectures in the automotive industry: Existing solutions, current problems and future challenges. http://2017.rtss.org/industrial-panel/.
[39] J. Reineke. Randomized caches considered harmful in hard real-time systems. *LITES*, 1(1), 2014.
[40] L. Santinelli et al. Revising measurement-based probabilistic timing analysis. In *RTAS*, 2017.
[41] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.
[42] Z. Stephenson et al. Supporting industrial use of probabilistic timing analysis with explicit argumentation. In *INDIN*, 2013.
[43] F. Wartel et al. Timing analysis of an avionics case study on complex hardware/software platforms. In *DATE*, 2015.

[44] Wilhelm R. et al. The worst-case execution-time problem overview of methods and survey of tools. *ACM TECS*, 7:1–53, May 2008.

[45] M. Ziccardi et al. EPC: extended path coverage for measurement-based probabilistic timing analysis. In *RTSS*, 2015.