A Cache Design for Probabilistically Analysable Real-time Systems

Leonidas Kosmidis[*,†], Jaume Abella[†], Eduardo Quiñones[†], Francisco J. Cazorla[†,‡]
[*]Universitat Politècnica de Catalunya
[†]Barcelona Supercomputing Center
[‡]Spanish National Research Council (IIIA-CSIC)

***Abstract*—Caches provide significant performance improvements, though their use in real-time industry is low because current WCET analysis tools require detailed knowledge of program's cache accesses to provide tight WCET estimates. Probabilistic Timing Analysis (PTA) has emerged as a solution to reduce the amount of information needed to provide tight WCET estimates, although it imposes new requirements on hardware design. At cache level, so far only fully-associative random-replacement caches have been proven to fulfill the needs of PTA, but they are expensive in size and energy.**

In this paper we propose a cache design that allows set-associative and direct-mapped caches to be analysed with PTA techniques. In particular we propose a novel *parametric random placement* suitable for PTA that is proven to have low hardware complexity and energy consumption while providing comparable performance to that of conventional modulo placement.

I. Introduction

Safety-critical real-time systems, such as flight control systems, have experienced an unprecedented growth in computational demands to cope with more sophisticated functionalities. This makes real-time designers to use processors with high performance features. One of the most used features to improve performance are cache memories. Unfortunately, the adoption of caches complicates the computation of worst-case execution time (WCET) estimates [9]. Several static methods have been devised to provide WCET estimates for systems with caches [12][18], but they require detailed knowledge of the sequence of cache accesses in order to provide tight WCET estimates. When the required knowledge is not available, pessimistic assumptions must be made by the analysis, resulting in overly pessimistic WCET estimates.

Recently, Probabilistic Timing Analysis (PTA) [7][6] has emerged as an alternative to conventional timing analysis. PTA provides WCET estimates with an associated probability of occurrence, called probabilistic WCET (pWCET) estimates. A pWCET estimate can be exceeded with a given probability, thus leading to a timing failure. This is analogous to the behaviour of hardware, for instance, which may fail with a given probability. In that sense, PTA extends the notion of probability of failure to timing correctness. To that end, PTA aims to obtain pWCET estimates for arbitrarily low probabilities, so that even if that pWCET estimate can be exceeded, it would be exceeded with low probability (e.g. in the region of 10^{-15} per hour of operation, largely below the probability of hardware failures). PTA can be applied either in a static (SPTA) [6] or measurement-based (MBPTA) [7] manner, as explained in Section II. SPTA derives a-priori probabilities for execution times from a model of a system, while the second variant, MBPTA, derives those probabilities by collecting observations of end-to-end runs of an application running on the target hardware.

PTA techniques, both SPTA and MBPTA, require that the observed execution times of programs have a distinct probability of occurrence and can be modelled with *independent and identically distributed* (i.i.d) variables. Unfortunately, these properties are not met by current processors due to their deterministic nature. For instance, if we run a given program fed with the same input data several times on the same processor, we will observe some variation in its execution time due to, for instance, the fact that it is allocated in different locations in memory resulting in different execution times. However, those execution times are not necessarily probabilistically modelable, i.e, cannot be modelled with random i.i.d variables.

At the cache level, the deterministic behaviour of the placement (e.g. modulo) and replacement (e.g. LRU) policies makes memory operations not to be modelable probabilistically. However, PTA has been shown to work with fully-associative (FA) caches deploying random replacement (RR) policy for both PTA variants, SPTA [6] and MBPTA [7]. FA-RR caches allow obtaining an actual hit/miss probability for each memory access, such that when a program runs several times on a processor with such a cache the obtained execution times are modelable with i.i.d random variables. Unfortunately, only small FA caches can be used in general due to their power hungry and costly implementation, thus constraining PTA applicability.

In this paper, we propose a new random placement policy that allows applying PTA methods not only to FA-RR caches but to set-associative and direct-mapped caches. While random replacement has been used in the past [2], [3], existing placement functions have a purely deterministic behaviour and thus, they cannot be used in the context of PTA because there is no way to determine the probability of each cache placement to occur at design time. To solve this problem, we propose a new cache design with *parametric random placement*, having the following properties: 1) The placement function is deterministic during the execution of the program enabling cache lookup to be performed analogously to deterministic-placement caches. However, placement is randomised across executions by modifying the seed of the parametric hash function used for placement. In this way, each memory access has hit/miss probabilities which leads to i.i.d. execution times as needed for PTA. 2) Our cache design has similar average performance to that of deterministic caches, which is important not to jeopardise other metrics such as energy consumption. Similar to FA-RR caches, our design also reduces drastically the amount of information required by the analysis to derive WCET estimates.

II. Background

State-of-the-art timing analysis techniques can be classified into two types [9]: Static timing analysis techniques construct a cycle-accurate model of the system and a mathematical representation of the code that are combined with linear programming techniques to determine a safe upper-bound on the WCET.

Measurement-based analysis techniques perform extensive testing on the real system under analysis using stressful, high-coverage input data, recording the longest observed execution time and adding to it an engineering margin to make safety allowances for the unknown. However, determining the engineering margin is extremely difficult — if at all possible — especially when the system may exhibit discontinuous changes in timing due to unanticipated timing behaviour.

PTA has emerged as an alternative to current timing analysis techniques. Both SPTA [6] and MBPTA [7] provide a cumulative distribution function, or pWCET function, that upper-bounds the execution time of the program under analysis, guaranteeing that the execution time of a program only exceeds the corresponding execution time bound with a probability lower than a given target probability (e.g., 10^{-15}). The probabilistic timing behaviour of a program (or an instruction) can be represented with Execution Time Profiles (ETPs). An ETP defines the different execution times of a program (or latencies of an instruction) and its associated probabilities. That is, the timing behaviour of a program/instruction can be defined by the pair of vectors $(\overrightarrow{l}, \overrightarrow{p}) = \{l_1, l_2, ..., l_k\}\{p_1, p_2, ..., p_k\}$, where p_i is the probability the program/instruction taking latency l_i. The ETP for a program (or instruction) may differ for different input sets leading to different execution paths. Each PTA technique has its own methods to combine results from different execution paths. We refer the reader to those methods for further details [6] [7].

A. Requirements of SPTA and MBPTA on Cache Design

PTA techniques require that the events under analysis, program execution times for MBPTA and instruction latencies for SPTA, can be modelled with *i.i.d.* random variables [6]: two random variables are said to be independent if they describe two events such that the occurrence of one event does not have any impact on the occurrence of the other event. Two random variables are said to be identically distributed if they have the same probability distribution.

The existence of an ETP ensures that each potential execution time of the program (for MBPTA) or instruction (for SPTA) have an actual probability of occurrence, which is a sufficient and necessary condition to achieve the desired probabilistic i.i.d. execution time behaviour [6].

A difference between SPTA and MBPTA, besides the level of abstraction at which ETPs are to be constructed, is that while SPTA requires ETPs for each instruction to be *determined*, MBPTA simply needs those ETPs for the program to *exist*, but not to be known.

Regardless of whether ETPs are obtained for instructions or full programs, they cannot be derived with current deterministic architectures since events affecting execution time, e.g. cache hits/misses, on those architectures cannot be attached a probability of occurrence. At the cache level, the problem resides on the deterministic behaviour of the placement and replacement policies, which (1) lead to cache layouts for which the corresponding execution times cannot be modelled with i.i.d. random variables preventing the use of MBPTA and (2) each memory request does not have an actual probability of hit/miss preventing its use with SPTA.

Overall, a SPTA- and MBPTA-analysable cache must provide the following properties:

a) SPTA: SPTA requires the i.i.d. hypothesis to strictly hold at the granularity level at which ETP are built, i.e. instructions. If the timing probability distribution captured by the ETP of the instruction is fully independent of the execution history, the ETP of the instruction would hold constant across all executions of the instruction. However, this is unaffordable at hardware level [6]. Instead, SPTA [6] also works with a SPTA-imperfect approach. In such approach the timing vector of the ETP is insensitive to execution history but the probability vector is not, and therefore, there is a need for bounding probabilistically this dependence. This PTA-imperfect approach provides safe pWCET estimates and is the one used in this paper. Hence SPTA requires that: 1) *Each memory access has a hit-miss probability*, and 2) *In case memory instructions are dependent, that dependence must be probabilistically modelable*.

b) MBPTA: The observed execution times fulfil the i.i.d. property if observations are independent across different runs and a probability can be attached to each potential execution time. To that end, it is enough if we *make the events that may affect the execution time of a program random events*. Hence, taking measurements from a program is equivalent to rolling a dice, with each face having a probability of appearance. Making enough rolls is enough to apply MBPTA, which derives upper-bounds of the execution time distribution by means of *Extreme Value Theory* (EVT) [11][7]. Note that *the existence of the ETPs for each instruction ensures that the execution times are probabilistic and therefore MBPTA can be applied*. As for SPTA, memory instructions may have dependences, but it is enough that those dependences are probabilistic, so that the measurements (execution times) obtained by running the program probabilistically capture the effect of such dependence.

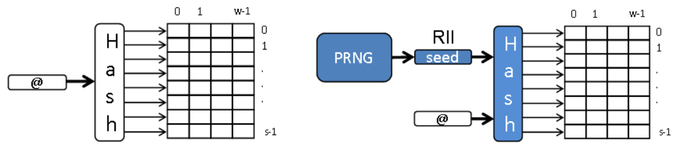
III. Timing Behaviour of Random Caches

This paper shows that randomising the replacement and placement policies allows constructing ETPs for memory instructions: $(\overrightarrow{l}, \overrightarrow{p}) = \{l_{hit}, l_{miss}\}\{p_{hit}, p_{miss}\}$, where l_{hit} and l_{miss} are the latency of hit and miss respectively and p_{hit} and p_{miss} the associated probability in each case. In particular, in this section, we show that p_{hit} and p_{miss} can be computed analytically based on the properties of RR and our random placement (RP) policy. As pointed out in Section II, the existence of the ETPs ensures that the execution times are probabilistic and therefore the system fulfils the i.i.d. properties.

A. Random Replacement (RR)

RR policy ensures that every time a memory request misses in cache, a way in its corresponding cache set is randomly selected and evicted to make room for the new cache line. This ensures that (1) there is independence across evictions and (2) the probability of a cache line to be evicted is the same across evictions, i.e. for a W-way associative cache, the probability for any particular cache line to be evicted is $\frac{1}{W}$ for each set. In the particular case of a fully-associative (FA) cache, such probability holds for the only cache set.

Given a sequence of cache accesses, the ETP for each of them (i.e. its hit/miss probabilities) can be determined by computing how likely previous accesses can evict the corresponding cache line. For instance, in the sequence <*A, B, C, A*>, B and C can evict A with a given probability that depends on the number of cache ways and whether B and C were fetched before or not. The fact that those probabilities exist and can be computed is enough for PTA techniques. Since cache lines evicted are chosen randomly, whether an access hits or misses depends solely on random events for a given sequence of accesses regardless of their absolute addresses, and thus hit/miss outcome is truly



(a) standard (b) proposed

Fig. 1. Block diagram of the cache design.

probabilistic. In particular, the hit probability (P_{hit}) of a given access A_j in the sequence $< A_i, B_{i+1}, ..., B_{j-1}, A_j >$, where A_i and A_j correspond to accesses to the same cache line and no B_k accesses cache line A, can be obtained as follows, with $P_{miss} = 1 - P_{hit}$ for any access:

$$P_{hit_{A_j}} = \left(\frac{W-1}{W}\right)^{\sum_{k=i+1}^{j-1} P_{miss_{B_k}}} \quad (1)$$

$P_{hit_{A_j}}$ is the probability of A surviving all evictions performed by $< B_{i+1}, ..., B_{j-1} >$. The probability of A to survive one random eviction is $\frac{W-1}{W}$. Meanwhile, given that one random eviction is performed on every miss, the total number of evictions equals the expected number of misses in between A_i and A_j, which is $\sum_{k=i+1}^{j-1} P_{miss_{B_k}}$. Using Equation (1) the hit/miss probabilities of each access can be derived sequentially starting from the first cache access. If no access has been performed to A before A_j, then the hit probability is zero.

Overall, the use of RR allows deriving an ETP for each memory operation, thus enabling PTA.

B. *Random Placement (RP)*

The RP policy we are after has to ensure that the cache set in which a cache line is mapped is randomly selected. Ideally, assuming a cache with S sets, the probability for a cache set to be selected is $\frac{1}{S}$.

One fundamental difference between placement and replacement policies is that placement assigns sets to cache lines based on the index bits of the memory address, Figure 1(a). As a result, if the placement policy assigns two memory addresses to the same cache set, they will collide systematically. To deal with this deterministic nature while randomising the timing behaviour of the placement policy, we propose a new parametric hash function that makes use of a random number as an input. Such random number can be generated either by hardware or software. Our hash function, given a memory address and a random number called *random index identifier* (RII), provides a unique and constant cache set (mapping) for the address along the execution, see Figure 1(b). If the RII changes, the cache set in which the address is mapped changes as well, so cache contents must be flushed for consistency purposes. We propose changing the RII only across program execution boundaries (e.g., the OS can do it) so that programs can be analysed with end-to-end runs without any further consideration than assuming that the cache is initially empty. We assume that given a memory address and a set of RIIs, the probability of mapping such address to a given cache set is the same, i.e. $\frac{1}{S}$, although this is not needed as long as such mapping is probabilistic. How we approximate this ideal distribution by hardware is shown in Section IV.

Next we describe how to quantify the probability of each memory address to be mapped into a given cache set, and so conflicting with other memory addresses. Given u different memory objects and S cache sets, we define *cache layout* as the resultant mapping of assigning the u memory objects into the S cache lines. Thus, every time the program is executed, a new RII is generated leading to a new random mapping function corresponding to a *cache layout*. Different cache layouts cause different cache conflicts among memory addresses, resulting in different execution times. The number of possible cache layouts is given by S^u, where S is the number of sets and u the number of distinct memory addresses.

Note that different cache layouts may be different but equivalent in terms of execution time. For instance, if we have three memory objects (A, B and C) and 4 cache sets, any cache layout where A is mapped in one cache set (e.g., set 0) and B and C in a different cache set (e.g., set 1) will be equivalent. Similarly, even non-equivalent cache layouts may lead to the same execution time. For instance, a particular cache layout may not have any conflict whereas another may create conflicts across addresses that would miss anyway. In both cases, although cache layouts are different, accesses will experience the same hits and misses.

By using a new RII on each execution, a random cache layout is chosen and pathological scenarios can only occur with a given probability.

In an arbitrary sequence $A, B_1, B_2, ...B_q, A$ where $\forall i,j : i \neq j$ and $B_i \neq B_j$, the probability of the second occurrence of A to survive (and so being a hit) in a direct-mapped cache is determined by those cache layouts in which the q objects in between are placed in a different cache set to A. If we consider that A is placed in a particular entry, the number of cache layouts in which the other q objects are placed in different cache sets is $(S-1)^q$: the q objects can be placed in all entries except where A is placed. Because A can be placed in any position, the number of cache layouts in which A survives is $(S-1)^q \cdot S$. Therefore, and considering that the number of possible cache layouts is determined by S^{q+1}, the probability of the second occurrence of A being a hit can be computed using the following equation:

$$P_{hit_A} = \frac{(S-1)^q \cdot S}{S^{q+1}} = \left(\frac{S-1}{S}\right)^q \quad (2)$$

The reuse distance of A, defined as the number of unique cache line addresses (q) between two occurrences of the same memory address, determines how likely it will result in a hit/miss. The higher the q-distance is between two occurrences, the less likely is the second occurrence of A to survive. For instance, A is more likely to be evicted in the sequence *A, B, C, A* ($q = 2$) than in the sequence *A, B, B, B, B, B, A* ($q = 1$).

Overall, the hit probability for any access exists (and so the miss probability). Therefore, the use of RP allows deriving an ETP for each memory operation, thus enabling PTA as it is the case for RR, because execution times will be i.i.d.

C. *Putting All Together: Set-Associative Caches*

The ETP of a memory operation accessing to a $S \cdot W$ set-associative cache with random placement and replacement policies is the combination of the ETPs of both policies. That is, the random placement will allocate memory objects into the S sets with a probability of $1/S$ while the random replacement policy will evict a way to allocate a new fetched cache line with a probability of $1/W$. In particular, an access A will miss only if a previous access misses in its same cache set and it randomly evicts its cache line. This can be formulated as follows:

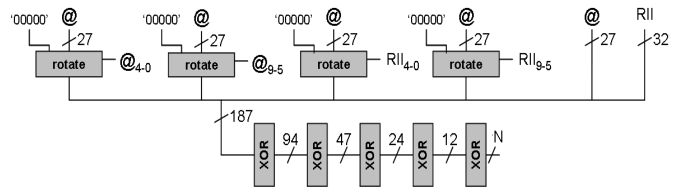


Fig. 2. Parametric hash function proposed for the RP cache.

$$P_{miss_A}(SA[S,W]) = P_{miss_A}(DM[S]) \cdot P_{miss_A}(FA[W]) \quad (3)$$

where $P_{miss_A}(SA[S,W])$ stands for the miss probability of A in a SA cache with S sets and W ways. Analogously, $P_{miss_A}(DM[S])$ and $P_{miss_A}(FA[W])$ stand for the miss probabilities in DM and FA caches with S sets and W ways respectively. Hit probabilities are obtained as $P_{hit} = 1 - P_{miss}$.

In summary, hit/miss probabilities exist for all accesses, and so their ETPs. As a consequence, execution times will be i.i.d. and PTA can be safely applied on top of a SA cache.

IV. Hardware Design of a Random Cache

This section describes how to implement both random placement and replacement policies.

A. Random Replacement

Random replacement policies have been extensively used in various processor architectures, both in the the high-performance and embedded markets. Examples for the latter market are the Aeroflex Gaisler NGMP [2] or some processors of the ARM family [3]. The most relevant element of a random replacement policy is the hardware generating random numbers which selects the way to be evicted on a miss. In general, pseudo-random number generators (PRNG) are implemented. The particular PRNG we have used in this paper is the Multiply-With-Carry (MWC) [14] PRNG, since we have tested that (i) it generates numbers with a sufficiently high level or randomness, (ii) its period is huge, and (iii) it can be efficiently implemented in hardware. Given that efficient implementations of a PRNG exist and space is limited, we omit the details of our implementation of the MWC PRNG.

B. Random Placement Policy

In this section, we propose an implementation of a random placement policy. The key components of this design are (1) a low-cost PRNG if the RII is produced by hardware and (2) a *parametric hash function*.

In order to keep cache latency and energy low, the implementation of both components must be kept simple. Moreover, both components are placed 'in front' of the cache, so the cache design is not changed per se, see Figure 1(b), but some extra logic is added before accessing cache. As for random replacement, we use the MWC PRNG if the RII is produced by hardware.

The Parametric Hash Function is used to randomise the cache placement. Figure 2 shows our implementation of the parametric placement function. The hash function has two inputs, the bits of the address used to access the set (index bits), '@' in the figure, and a RII. In the configuration of the particular example, 32 bytes per cache line and 32-bit addresses are assumed. Therefore, the 5 lowermost bits are discarded (offset bit) and only 27 bits are used.

The hash function rotates the address bits, based on some bits of the RII as it is shown in the two rightmost rotate blocks of the figure. By doing this, we ensure that when a different RII is used, the mapping of that address changes. Analogously, the address bits are rotated based on some bits of the address itself. This operation, which is performed by the two leftmost rotate blocks, changes the way that the addresses are shifted. Note that addresses are padded with zeros to obtain a power-of-two number of bits, so address bits can be rotated without any constraint. Otherwise, rotation values between 27 and 31 would require special treatment.

Finally, all bits of the rotated addresses, the original address and the RII (187 bits in the example), are XORed successively, until we obtain the desired number of bits for indexing the cache sets. For example, a 16KB cache with 32 bytes per line would need 9 index bits for a direct-mapped organisation, 8 bits for a 2-way set-associative, and so on and so forth. Hence, 5 XOR gate levels are enough to produce the index.

As shown in Figure 2, the hardware implementation of the hash function consists of 4 rotate blocks and 5 levels of 2-input XOR gates. Each rotate block can be implemented with a 5-level multiplexer [19]. Since the latency and the energy per access of a fully-associative cache is much larger than the one of direct-mapped or set-associative caches, the relative overhead of the hash function is small. We have corroborated this observation by integrating our parametric placement function into the CACTI tool [15]. Results for several cache configurations show that energy per access grows around 3% and delay grows by 40% (it is still less than half the delay of a fully-associative cache). Note that hit latency has low impact in WCET since it is typically some orders of magnitude lower than miss latency. Nevertheless, we assume the same hit latency for our DM and SA configurations, and the FA one, which plays against our proposal.

V. Results

A. Experimental Setup

We use a cycle-accurate execution-driven simulator based on the SoCLib simulation framework [21], with PowerPC binaries [23]. The simulator models a 4-stage pipelined processor with a memory hierarchy composed of first level separated instruction and data caches, and main memory. Both instruction and data cache size is 4-KB with 16-byte line size. Associativities considered are 1-way (direct-mapped), 8-way (set-associative), 32-way (set-associative) and 256-way (fully-associative). Both caches implement random replacement and our random placement policy.

The latency of the fetch stage depends on whether the access hits or misses in the instruction cache: a hit has 1-cycle latency and a miss has 100-cycle latency. After the decode stage, memory operations access the data cache so they can last 1 or 100 cycles depending on whether they miss or not. The remaining operations have a fixed execution latency (e.g. integer additions take 1 cycle).

We use the EEMBC Autobench benchmark suite [16] that reflects the current real-world demand of some automotive critical real-time embedded systems.

B. Fulfilling the i.i.d properties

The use of random replacement and placement policies guarantees that observed execution times fulfil the properties required by MBPTA. However, we further verify this

TABLE I
INDEPENDENCE AND IDENTICAL DISTRIBUTION TESTS RESULTS (OUTCOME INDEPENDENCE TEST / OUTCOME I.D TEST)

Benchmarks	**1w-256s DM**	**8w-32s SA**	**32w-8s SA**	**256w-1s FA**
a2time	0.63 / 0.43	0.90 / 0.64	0.88 / 0.75	0.53 / 0.57
aifftr	0.07 / 0.98	0.01 / 0.93	0.74 / 0.92	0.59 / 0.74
aifirf	0.39 / 0.83	0.27 / 0.84	0.76 / 0.28	0.21 / 0.36
aiifft	0.23 / 0.81	0.11 / 0.70	0.19 / 0.53	0.05 / 0.34
cacheb	0.53 / 0.48	0.51 / 0.40	0.27 / 0.97	1.20 / 0.36
canrdr	0.17 / 0.53	0.21 / 0.39	1.27 / 0.12	0.23 / 0.29
iirflt	1.27 / 0.84	0.11 / 0.80	0.05 / 0.49	0.71 / 0.75
puwmod	0.17 / 0.47	0.37 / 0.89	0.41 / 0.96	0.51 / 0.92
rspeed	0.33 / 0.89	0.33 / 0.27	0.25 / 0.24	1.24 / 0.60
tblook	0.51 / 0.35	0.47 / 0.93	0.67 / 0.91	0.03 / 0.54
ttsprk	0.82 / 0.13	0.63 / 0.73	0.19 / 0.92	1.12 / 0.80

point empirically by analysing whether execution times of EEMBC benchmarks on 4 different cache configurations (see Section V-A) are independent and identically distributed.

In order to test independence we use the Wald-Wolfowitz independence test [5]. We use a 5% significance level (a typical value for this type of tests), which means that absolute values obtained after running this test is lower than 1.96 if there is independence, and higher otherwise. For identical distribution, we use the two-sample Kolmogorov-Smirnov identical distribution test [4] as described in [7]. For 5% significance, the outcome provided by the test should be above the threshold (0.05) to indicate identical distribution, and non-identical distribution otherwise.

Table I shows the results of both tests for all EEMBC benchmarks under cache configurations varying the number of ways (*w*) and sets (*s*), when running each benchmark 1,000 times. As shown, both tests are passed in all cases.

C. Performance Analysis

Next, we compare the average performance of deterministic and random caches. In particular, we compare different *random placement+replacement* caches against *modulo placement and LRU replacement* caches for different associativities. Table II shows the average IPC (instructions per cycle) for all EEMBC benchmarks under different cache configurations and 1,000 runs per benchmark. Execution time variation is quite low except for direct-mapped caches where some mappings are particularly good (such as those for modulo placement) whereas others create some conflicts.

If we focus on the DM-RP cache, we observe that, as expected, it performs much worse than the FA one due to the reduced associativity. Further, the DM-RP cache performs much worse than the DM-modulo cache. This is so because memory objects (for both code and data) are laid out sequentially and hence, modulo placement minimises conflicts. Conversely, even if random placement creates few conflicts, those hold for the whole execution. On average, random placement does not introduce an excessive number of misses, but their impact in the IPC is relevant due to the large miss latency (100 cycles).

For the SA+RP-RR configurations we observe that few number of ways is enough to provide average performance close to that of the FA-RR cache and to that of LRU+modulo caches. In particular, 8- and 32-way SA-RP+RR cache designs provide performance comparable to state-of-the-art cache designs (12% and 10% slowdown respectively), and acceptable design cost (largely below unaffordable large fully-associative caches) while enabling PTA.

TABLE II
IPC OF RP+RR AND MODULO+LRU CACHES IN OUR ARCHITECTURE

	1w-256s DM	**8w-32s SA**	**32w-8s SA**	**256w-1s FA**
RP+RR	0.234	0.585	0.615	0.627
LRU+modulo	0.613	0.665	0.687	0.698

D. MBPTA: EVT projections

In this section we provide some pWCET estimates obtained with the method provided in [7]. Note that MBPTA has been used so far *only* on top of FA-RR caches. Although FA-RR caches accomplish the properties required by PTA, they have high hardware implementation cost and low scalability. Therefore, this paper provides the first SA and DM cache designs amenable for PTA.

Following the iterative method of [7] we carried out 1,000 experiments and use EVT to extract pWCET estimates. Figure 3 shows the EVT projections generated with [7] of `aifftr` considering a FA-RR cache (labelled as *1s - 256w*), two SA-RP+RR caches (labelled as *8s - 32w* and *32s - 8w*) and a DM-RP cache (labelled as *256s - 1w*). As expected, the FA-RR cache provides the lowest pWCET estimates. That is, the random replacement policy has lower probability of resulting in cache layouts with multiple cache conflicts because random choices are taken on every miss instead of across different runs. However, as we reduce the associativity of the cache, and so we increase the number of sets, the number of cache layouts decreases, thus increasing the probability of having more cache conflicts.

The pWCET increment due to the reduction of the cache associativity depends on the application: for instance, *a2time* is very sensitive to cache associativity. Instead, *aifftr* experiences pWCET estimate increments of only 9% and 11% when considering SA-RP+RR caches and up to 5x when considering a DM-RP cache, with respect to the FA-RR cache.

Table III shows the pWCET increment of the SA-RP+RR and DM-RP caches with respect to the FA-RR cache for all benchmarks when considering an exceedance probability of 10^{-13}. Our selection of the exceedance probability, i.e. the probability that an instance of a task misses its deadline, is based on the observation that for the aerospace commercial industry at the highest integrity level DAL-A the maximum allowed failure rate in a piece of software is 10^{-9} per hour of operation [1]. In current implementations, the highest frequency at which a task can be released is 20 milliseconds (2×10^{-3}) [1]. Hence, the highest allowed failure rate per task activation is 2×10^{-11}, which is largely above our exceedance probability. Nevertheless, similar trends are observed for other exceedance probabilities. Overall, our RP+RR cache designs provide the best tradeoff between hardware complexity and pWCET for PTA while not requiring information about the actual addresses accessed by the programs analysed. Moreover, second level caches can be used to mitigate the large impact of misses in both average performance and pWCET estimates.

VI. RELATED WORK

Caches pose serious challenges on WCET analysis methods. In fact, cache WCET impact has been studied extensively by the research community [18], including several levels of cache [12] and locking mechanisms [17] to increase predictability and hence provide tighter WCET estimations.

Some current processors used in high-performance embedded systems already implement random replacement policies

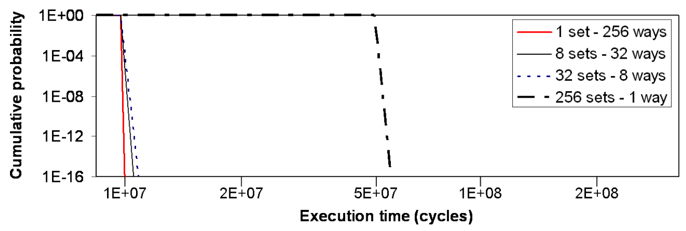


Fig. 3. EVT projection for *aifftr*

TABLE III
pWCET INCREMENT OF THE SA AND DM CACHES WITH RESPECT TO THE FA ONE, CONSIDERING AN EXCEEDANCE PROBABILITY OF 10^{-13}

Benchmarks	32w-8s (SA)	8w-32s (SA)	1w-256s (DM)
a2time	452%	511%	1758%
aifftr	9%	11%	468%
aifirf	61%	65%	1418%
aiifft	9%	12%	653%
cacheb	9%	12%	904%
canrdr	8%	9%	2126%
iirflt	370%	478%	1448%
puwmod	29%	31%	855%
rspeed	23%	25%	12691%
tblook	167%	185%	1995%
ttsprk	13%	14%	3386%

on set-associative caches [3][2]. Randomised caches in high-performance processors have been proposed to remove cache conflicts by using pseudo-random hash functions [22][10][20]. However, the behaviour of all those cache designs is fully deterministic, and therefore, whenever a given input set produces a pathological access pattern, it will happen systematically for such input set. Therefore, although the frequency of pathological cases is reduced, they can still appear systematically because there is no way to prove that their probability is bound.

Some work on PTA has been done based on the assumption that execution times are truly i.i.d. and that frequencies for execution paths provided by the user match actual probabilities of those paths [8]. Later work has shown how to perform PTA with no assumption on the probabilities of execution paths and how to use random caches in PTA systems [6][7]. Concretely, authors showed that randomised replacement effectively avoids pathological behaviour of deterministic replacement policies while achieving reasonable performance. Some authors have tried to perform PTA on top of conventional cache designs [13]. Unfortunately, this can only be done if the user is able to provide the *true probability* (not the frequency) of each cache layout and each execution path to occur for *all instances* of the system deployed, which is, in general, unattainable.

To the best of our knowledge, our paper is the first enabling the use of the most common and efficient cache designs, i.e. set-associative and direct-mapped caches in probabilistically analysable hard real-time systems while preserving the properties needed by sound PTA techniques [6][7].

VII. Conclusions and Future Work

PTA enables affordable analysis of complex hardware in safety-critical real-time systems by reducing the amount of information about the hardware and software state required to provide trustworthy WCET estimates. Yet, PTA relies on some properties that existing hardware fails to provide. In particular PTA requires that the execution times of the program on the target platform can be modelled with i.i.d random variables.

In the case of the cache, the deterministic behaviour of placement and replacement policies makes it impossible to assign a true probability to different execution times. Only unaffordable fully-associative caches with random replacement would allow deriving true probabilities. This paper presents the first random placement policy based on a parametric hash function so that i.i.d. execution times are achieved, thus enabling the use of efficient set-associative and direct-mapped caches in the context of probabilistic timing analysis. We further show that our cache design can be implemented with little overhead in terms of complexity, energy and performance.

While in this paper we have focused on devising random placement and replacement policies and implementations for first level caches, we plan to extend random placement policies to other components such as second level caches and translation look-aside buffers (TLBs).

Acknowledgments

This work has been supported by the PROARTIS FP7 European Project under grant agreement number 249100, the Spanish Ministry of Science and Innovation under grant TIN2012-34557 and the HiPEAC Network of Excellence. Leonidas Kosmidis is also funded by the Spanish Ministry of Education under the FPU grant AP2010-4208. Eduardo Quiñones is also funded by the Spanish Ministry of Science and Innovation under the Juan de la Cierva grant JCI2009-05455.

References

[1] Guidelines and methods for conducting the safety assessment process on civil airborne systems and equipment. *ARP4761*, 2001.
[2] Aeroflex Gaisler. *Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual*, 2011.
[3] ARM. *Cortex-R4 and Cortex-R4F Technical Reference Manual*, 2006.
[4] Sarah Boslaugh and Paul Andrew Watters. *Statistics in a nutshell*. O'Reilly Media, Inc., 2008.
[5] J.V. Bradley. *Distribution-Free Statistical Tests*. Prentice-Hall, 1968.
[6] F.J. Cazorla, E. Qui nones, T. Vardanega, L. Cucu, B. Triquet, G. Bernat, E. Berger, J. Abella, F. Wartel, M. Houston, L. Santinelli, L. Kosmidis, C. Lo, and D. Maxim. Proartis: Probabilistically analysable real-time systems. Technical Report 7869(http://hal.inria.fr/hal-00663329), INRIA, to appear in ACM TECS, 2012.
[7] L. Cucu, L. Santinelli, M. Houston, C. Lo, T. Vardanega, L. Kosmidis, J. Abella, E. Mezzeti, E. Qui nones, and F.J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *ECRTS*, 2012.
[8] Laurent David and Isabelle Puaut. Static determination of probabilistic execution times. In *ECRTS*, 2004.
[9] R. Wilhelm et al. The worst-case execution time problem: overview of methods and survey of tools. *Trans. on Embedded Computing Systems*, 7(3):1–53, 2008.
[10] A. González et al. Eliminating cache conflict misses through XOR-based placement functions. In *ICS*, 1997.
[11] Samuel Kotz and Saralees Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.
[12] Benjamin Lesage, Damien Hardy, and Isabelle Puaut. Wcet analysis of multi-level set-associative data caches. *WCET Workshop*, 2009.
[13] Yun Liang and Tulika Mitra. Cache modeling in probabilistic execution time analysis. In *DAC*, 2008.
[14] G. Marsaglia and A. Zaman. A new class of random number generators. *Annals of Applied Probability*, 1(3):462–480, 1991.
[15] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. CACTI 6.0: A tool to understand large caches. *HP Tech Report HPL-2009-85*, 2009.
[16] Jason Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[17] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS*, 2002.
[18] J. Reineke et al. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, November 2007.
[19] S. Huntzicker et al. Energy-delay tradeoffs in 32-bit static shifter designs. In *ICCD*, 2008.
[20] A. Seznec and F. Bodin. Skewed-associative caches. In *PARLE*. 1993.
[21] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.
[22] Nigel Topham and Antonio González. Randomized cache placement for eliminating conflicts. *IEEE Trans. Comput.*, 48, February 1999.
[23] J. Wetzel, E. Silha, C. May, B. Frey, J. Furukawa, and G. Frazier. *PowerPC User Instruction Set Architecture*. IBM Corporation, 2005.