Efficient Cache Designs for Probabilistically Analysable Real-time Systems

Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones and Francisco J. Cazorla

Abstract—The increasing performance demand in the critical real-time embedded systems (CRTES) domain calls for high-performance features such as cache memories. Unfortunately, the cost to provide trustworthy and tight Worst-Case Execution Time (WCET) estimates in the presence of caches is high with current practice WCET analysis tools because they need detailed knowledge of program's cache accesses to provide tight WCET estimates. The advent of Probabilistic timing analysis (PTA) opens the door to economically-viable timing analysis in the presence of caches, but 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 their energy, delay and area cost is unaffordable for CRTES.
In this paper we propose the first PTA-compliant cache design based on set-associative and direct-mapped arrangements, as those are the most common arrangements. In particular we propose a novel *parametric random placement* policy suitable for PTA that is proven to have low hardware complexity and energy consumption while providing comparable performance to that of conventional modulo placement.

Index Terms—Cache memories, Worst-case analysis

1 Introduction

Embedded real-time industries, such as avionics, automotive, railway and space, have steadily growing demands for greater computing power and stricter cost containment [8]. Real-time system designers respond to this need by using processors with high-performance features. In particular, cache memories have been shown to be a key feature to improve average performance. Unfortunately, their adoption challenges the computation of worst-case execution time (WCET) estimates [28], which is mandatory in real-time systems to prove that the timing constraints of the system are met.

Several static timing analysis methods have been devised to provide WCET estimates for systems with caches [15][21], but they require detailed knowledge of the sequence of memory addresses accessed to provide tight WCET estimates. Often such knowledge is not fully available due to the use of software features such as pointers. In those cases, the lack of information on the addresses that will access the cache forces analysis techniques to make pessimistic assumptions, resulting in pessimistic WCET estimates. Alternatively, programming guidelines facilitating the determination of addresses can be followed (e.g., avoiding the use of pointers), but they come at the expense of lower performance, lower maintainability and lower portability of software.

Measurement-based timing analysis methods can also be used in the presence of caches [28]. Those methods rely on executing the program with stressful tests so that the highest execution time observed plus an engineering margin can be used as WCET estimate. Regrettably, determining such margin is an open problem and strong guarantees needed in CRTES may not be met. This issue exacerbates in the presence of cache memories, since small variations in the program input can cause abrupt performance variations, thus threating those margins derived during timing analysis.

Recently, Probabilistic Timing Analysis (PTA) [9][7] has emerged as an alternative to conventional timing analysis. PTA provides WCET estimates with an associated exceedance probability, called probabilistic WCET (pWCET) estimates. A pWCET estimate can be exceeded with a given probability, thus leading to a timing failure. This is similar 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 at obtaining 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) [7] or measurement-based (MBPTA) [9] manner. SPTA derives a-priori probabilities for execution times of

- *L. Kosmidis is with the Universitat Politècnica de Catalunya (UPC) and Barcelona Supercomputing Center (BSC-CNS). E-mail: leonidas.kosmidis@bsc.es*
- *J. Abella is with the BSC-CNS. E-mail: jaume.abella@bsc.es*
- *E. Quiñones is with the BSC-CNS. E-mail: eduardo.quinones@bsc.es*
- *F.J. Cazorla is with the BSC-CNS and Spanish National Research Council (IIIA-CSIC). E-mail: francisco.cazorla@bsc.es*

Manuscript received 27 Feb. 2013; revised 28 Aug. 2013; accepted 2 Sep. 2013; published online xx XXX. 2013.
Recommended for acceptance by X. XXXXXX.
For information on obtaining reprints of this article, please send e-mail to: tc@computer.org, and reference IEEECS Log Number TC-2013-02-0146.
Digital Object Identifier no. 10.1109/TC.2013.XXX

each instruction, from a model of a system, that are combined to derive a pWCET for the program. MBPTA, instead, derives those probabilities by collecting observations of end-to-end runs of a program running on the target hardware. While MBPTA requires that the observed execution times of programs have a distinct probability of occurrence and can be modelled with random *independent and identically distributed* (i.i.d) variables, SPTA requires that the execution time of each instruction can be modelled with i.i.d random 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 modellable, 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., least recently used or LRU for short) policies makes memory operations not to be modellable probabilistically. However, PTA has been shown to work with fully-associative (FA) caches deploying random replacement (RR) policy for both PTA variants, SPTA [7] and MBPTA [9]. 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 modellable with i.i.d random variables. Furthermore, execution time variations due to minor changes in the input set of programs do not create abrupt execution time variations. 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 implementing a *parametric random placement* function with 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.
2) 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 so that execution times attain i.i.d properties as needed for PTA.
3) Our cache design has similar average performance to that of deterministic caches, which is important not to jeopardise other metrics such as energy consumption.
4) Similarly to FA-RR caches, our design also reduces drastically the amount of information required by the analysis to derive WCET estimates since knowing the memory addresses accessed is not needed anymore.

The rest of this paper is organised as follows. Section 2 provides background on timing analysis and their requirements on the cache design. Sections 3 and 4 introduce our new cache designs and their implementation. Section 5 proves that our designs are suitable for PTA and evaluates their performance. Section 6 presents some related work. Section 7 draws the main conclusions of this work.

2 Background

This work targets critical real-time embedded systems (CRTES). CRTES main requirements are on the functional and timing behaviour of the system. Functional guarantees are achieved by means of thorough validation and verification processes. Timing guarantees – the main focus of this work – require estimating the WCET of programs executing critical functions (e.g., break control in a car, flight control systems in a plane) so that it can be proven that those functions will execute *timely* in the system once deployed. Average performance and energy consumption, primary metrics for high-performance systems, are second-order problems in comparison with providing timing guarantees. The WCET estimation process requires obtaining trustworthy and tight WCET estimates so that strong guarantees can be provided, as needed by safety-related standards and WCET bounds are still close enough to reality so that computation resources are not wasted when scheduling programs. In this section we provide some background on timing analysis methods and their requirements on cache design.

2.1 Timing Analysis Methods

Static timing analysis techniques [28] 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 [28] 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 [7] and MBPTA [9], though their functioning is different, 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 $(\vec{l}, \vec{p}) = \{l_1, l_2, ..., l_k\}\{p_1, p_2, ..., p_k\}$, where p_i is the probability the program/instruction taking latency l_i, with $\sum_{i=1}^{k} p_i = 1$.

The ETP for a program (or instruction) for a given input set leading to a single execution path is obtained as follows:

SPTA In SPTA, ETPs for individual instructions are determined statically from a model of the processor and software. SPTA is performed by calculating the *convolution* of the discrete probability distributions which describe the execution time for each instruction; this provides a probability distribution, or ETP, representing the timing behaviour of the entire sequence of instructions. For instance the convolution of $ETP_1 = \{\{1,7\},\{0.4,0.6\}\}$ and $ETP_2 = \{\{2,4\},\{0.5,0.5\}\}$ consists in adding the timing vectors and multiplying the probability vectors, resulting in $ETP_3 = \{\{3,5,9,11\},\{0.2,0.2,0.3,0.3\}\}$.

MBPTA Given a set of R runs of a program, one could compute the pWCET function of the program as the *exceedance cumulative distribution function* (ECDF). ECDF provides the probability of occurrence of each of the observed execution times based on the histogram of execution times. Unfortunately, ECDF can only provide execution time estimates for probabilities down to $\frac{1}{R}$ in the best case. For smaller probabilities, techniques such as *Extreme Value Theory* (EVT) [14][9] are used to project an upper-bound of the tail of the exceedance function, enabling MBPTA techniques to provide pWCET estimates for *target probabilities* largely below $\frac{1}{R}$.

ETP 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 [7] [9] since those methods do not impose any further requirement on the underlying hardware.

2.2 SPTA/MBPTA Requirements 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 [7]: 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) has an actual probability of occurrence, which is a sufficient and necessary condition to achieve the desired probabilistic i.i.d. execution time behaviour.

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 or its components (e.g., instructions) 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:

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 complicates SPTA because different execution paths and even the outcome of the random events in a single path alter the probability vector. SPTA, as presented in [7], advocates for doing per-path WCET analysis which are then combined to obtain the WCET estimate for the program. Within a path, 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. Hence SPTA requires that: 1) *Each memory access has a hit-miss probability*, and 2) *If memory instructions are dependent, that dependence must be probabilistically modellable.*

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 occurrence. Making enough rolls of the different paths relevant for WCET is enough to apply MBPTA, which derives upper-bounds of the execution time distribution by means of *Extreme Value Theory* (EVT) [14][9]. 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

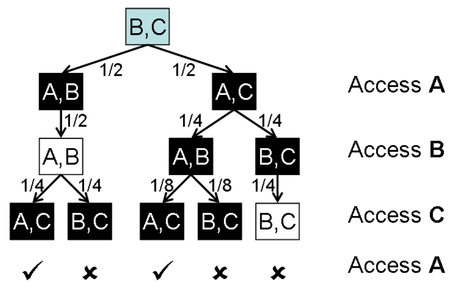

Fig. 1. Probability tree of the sequence $< A, B, C, A >$. Each box represents the cache state after a given access. Black boxes indicate that the current access misses in cache, while white boxes indicate that the current access hits.

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.

3 Timing Behaviour of Random Caches

This paper shows that randomising the replacement and placement policies allows constructing ETPs for memory instructions: $(\vec{l}, \vec{p}) = \{l_{hit}, l_{miss}\}\{p_{hit}, p_{miss}\}$, 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 2, the existence of the ETPs ensures that the execution times are probabilistic and therefore the system fulfills the i.i.d. properties.

3.1 Random Replacement (RR)

RR policy ensures that every time a memory request misses in cache, a way in the 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. In order to illustrate this, let us assume a FA cache with two ways $W = 2$ that initially contains B and C, each one in a different way. Figure 1 shows all possible cache states with their associated probabilities after executing each access in the sequence *<A, B, C, A>*. Black boxes represent cache states in which a miss occurs, while white boxes represent cache states in which a hit occurs. For instance, if the first access to A evicts C (leftmost branch in the tree), B survives and the access B hits in cache. In that case, the next access to C will miss in cache and may or may not evict A (a similar reasoning can be followed for the other subtrees). Overall, the second occurrence of A will hit in cache if and only if the replacement policy does not evict A when B and C are accessed. This means that the second occurrence of A has a hit probability of $\frac{3}{8}$.

Computing hit probabilities for an arbitrarily large number of cache accesses would be costly (but doable) with this method. The fact that those probabilities exist and can be computed is enough for PTA techniques to be applied. In the particular case of MBPTA, the existence of those probabilities [9] is enough, but there is no need for computing them.

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 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 approximated as follows, with $P_{miss} = 1 - P_{hit}$ for any access:

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

$P_{hit_{A_j}}$ is the probability of A surviving all evictions performed by $< B_{i+1}, ..., B_{j-1} >$, which depends on the probability of surviving one eviction times the number of evictions in that sequence. 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 $\sum_{k=i+1}^{j-1} P_{miss_{B_k}}$. In the worst case $P_{miss_{B_k}} = 1$ and so $\sum_{k=i+1}^{j-1} P_{miss_{B_k}} = (j - i - 1)$.

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.

Cache layouts: In this subsection, we further elaborate on the distinct number of cache states during the execution of a sequence of accesses, and introduce the concept of *cache layout* as a means to link random replacement and random placement caches.

On every access to a FA cache, an associative search is done among all the different cache ways. On an eviction, the new fetched cache line can be placed in

TABLE 1
Cache layouts for the sequence <A, B, C, A> in a 2-way FA cache

Way 1	**Way 2**
A_1, B, C, A_2	
A_1, B, C	A_2
A_1, B, A_2	C
A_1, C, A_2	B
B, C, A_2	A_1
A_1, B	C, A_2
A_1, C	B, A_2
A_1, A_2	B, C

any cache way. We define *cache layout* as the resultant address-to-cache-line mapping after assigning one or several memory objects into the cache. In the case of a FA cache, a new *cache layout* is built at every cache miss because, on every miss the newly fetched address (object) is placed in a random line.

This makes, for a given sequence of memory addresses, the number of cache layouts depend on the number of evictions happening in that sequence, which is bounded by the number of cache accesses in the sequence (m). For instance, $m = 4$ in our previous example (<A, B, C, A>). The total number of cache layouts is thus upper-bounded by the m^{th} *moment of a probability distribution of random permutations* [11]:

$$E(X^m) = \sum_{j=0}^{W} S_{m,j} \tag{2}$$

where X stands for the random variable that models the cache behaviour, i.e. the random replacement policy, and $S_{m,j}$ stands for the Stirling number of the second kind with parameters m and j [6]. The m^{th} moment is the number of partitions of a set of cache accesses (m) into no more than W cache ways. In our example, $E(X^m) = 8$. Table 1 provides the 8 different cache layouts for our example, where A_1 and A_2 stand for the first and second occurrence of A respectively. Note that the order is not relevant for cache layouts since, for instance, mapping A_1, B and C in the first cache line and A_2 in the second line is analogous to mapping A_2 in the first cache line and A_1, B and C in the second line.

3.2 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 2(a). As a result, if the placement policy assigns two memory addresses to the same cache set, they will collide systematically during the entire execution of the program. 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

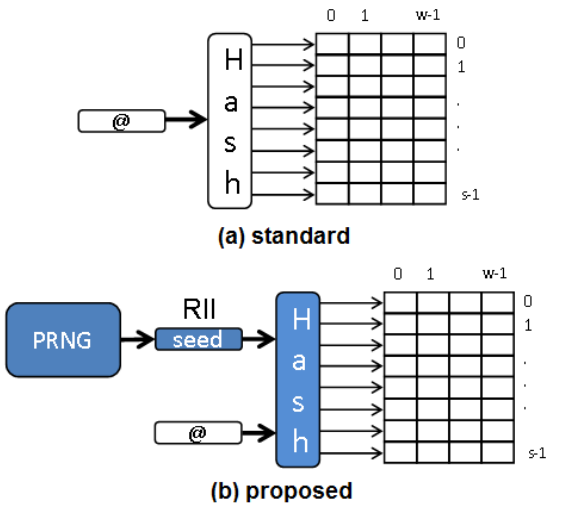

Fig. 2. Block diagram of the cache design.

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 2(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 so that programs can be analysed with end-to-end runs without any further consideration than assuming that the cache is initially empty[1]. 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 4.

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, a new *cache layout* results when the placement policy assigns (maps) the u memory objects into the S cache sets. Every time the program is executed, a new RII is generated leading to a new random mapping function corresponding to a new *cache layout*.

Note that different cache layouts cause different cache conflicts among memory addresses, resulting in different execution times. Further note that different cache layouts may lead to the same execution time. For instance, if we have three memory objects (A, B and C) and 4 cache sets, any cache layout where A is

1. The RII could also be changed periodically as a means to increase the degree of randomisation; however, it would be necessary to flush caches. The maximum execution time impact of this process (cache flushing and serving as many misses as cache lines) should be then accounted for. As shown in [26], this is not needed for real applications as the degree of randomisation achieved by changing the RII across execution boundaries is enough to obtain WCET estimates close to the actual performance on conventional caches.

TABLE 2
Possible cache layouts for the different accesses of the sequence <*A*, *B*, *C*, *A*> in a idealised random placement cache with two sets.

Cache layout	Conflicts (id)	P_{layout}
$A_0B_0C_0$	*A*, *B* and *C* (1)	$(\frac{1}{2})^3$
$A_0B_0C_1$	*A* and *B* (2)	$(\frac{1}{2})^2(1-\frac{1}{2})$
$A_0B_1C_0$	*A* and *C* (3)	$(\frac{1}{2})^2(1-\frac{1}{2})$
$A_0B_1C_1$	*B* and *C* (4)	$(1-\frac{1}{2})^3$
$A_1B_0C_0$	*B* and *C* (4)	$(1-\frac{1}{2})^3$
$A_1B_0C_1$	*A* and *C* (3)	$(\frac{1}{2})^2(1-\frac{1}{2})$
$A_1B_1C_0$	*A* and *B* (2)	$(\frac{1}{2})^2(1-\frac{1}{2})$
$A_1B_1C_1$	*A*, *B* and *C* (1)	$(\frac{1}{2})^3$

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 in terms of execution time.

To develop our argument let us assume a random placement direct-mapped cache composed of $S = 2$ cache sets, where no replacement policy is needed and hence, only the placement determines the cache layout and so the execution time. Table 2 identifies all possible cache layouts of a program consisting of $u = 3$ memory objects mapping into different cache lines (<*A*, *B*, *C*, *A*>). The subscript in each address in the first column indicates the cache set on which each address is mapped, 0 or 1 in this example.

With random placement we can derive the probability of each cache layout to occur. The column labelled as P_{layout} in Table 2 shows the probability of each cache layout to happen: The probability of the cache layout in which *A*, *B* and *C* are mapped into the same set ($A_0B_0C_0$ and $A_1B_1C_1$) is $(\frac{1}{2})^3$ each. Similarly, the probability of the cache layout in which A is mapped in a different entry to *B* and *C* ($A_0B_1C_1$ and $A_1B_0C_0$) is $(1-\frac{1}{2})^3$.

However, we are not interested in all cache layouts, but only in those that may produce different execution times. For instance, in Table 2, cache layouts $A_0B_0C_1$ and $A_1B_1C_0$ result in exactly the same cache conflicts (and so the same execution time), because they will experience exactly the same misses under both cache layouts. The total number of cache conflict layouts is given by the u^{th} *moment of a probability distribution of random permutations* [11]:

$$E(X^u) = \sum_{j=0}^{S} S_{u,j} \tag{3}$$

The u^{th} moment is the number of partitions of u unique memory addresses into no more than S cache sets. Thus, $E(X^u)$ provides the number of unique cache conflicts among the u memory addresses. In the example above, the number of possible cache layouts is 4, identified by a number in parenthesis in the second column of Table 2.

With this, we can compute the probability of each cache layout and hence the probability of its resulting execution time for a given program. If we consider the example shown in Table 2, and we assume a hit latency of 1 cycle and a miss latency of 10 cycles, we can derive the following probability distribution function for the observed execution times: $\{(31, 40), (0.25, 0.75)\}$. Cache layouts (1), (2) and (3) lead to an execution time of 40 cycles with an associated probability of 0.25 each. Cache layout (4) leads to an execution time of 31 cycles with an associated probability of 0.25. Note that still non-equivalent cache layouts may lead to the same execution time as it is the case for cache layouts (1), (2) and (3) depending on how accesses to the u memory objects interleave. Thus, analogously to $E(X^m)$ for random-replacement caches, $E(X^u)$ is an upper bound of the number of cache layouts for random-placement caches.

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

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 set, 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 sets except the one 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 S^{q+1}, the probability of the second occurrence of A being a hit can be approximated using the following equation:

$$P_{hit_A} = \frac{(S-1)^q \cdot S}{S^{q+1}} = \left(\frac{S-1}{S}\right)^q \tag{4}$$

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 A will result in a hit/miss. The higher the q-distance is between two occurrences of A, the less likely 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.

3.3 Putting All Together: Set-Associative Caches

The number of cache layouts for a set-associative cache implementing random placement and replacement can be computed as the combination of the number of cache layouts provided by the placement and replacement policies. Thus, the number of cache conflict layouts can be computed as the product of the

u^{th} and m^{th} moments of a probability distribution of random permutations:

$$E(X^m) \cdot E(X^u) = \sum_{j=0}^{W} S_{m,j} \cdot \sum_{j=0}^{S} S_{u,j} \tag{5}$$

$E(X^u)$ and $E(X^m)$ are the number of cache layouts given by the random placement and random replacement policies respectively.

For example, if we consider a program composed of the sequence of memory accesses *<A, B, C, D, A, B, A, C, A, D, A, B, A, C, A, D>*, in which $u = 4$ and $m = 16$, and a cache with 4 cache lines, we can compute the number of cache layouts for different cache configurations. In particular, for this example, we consider a direct-mapped random-placement (DM-RP) cache with $S = 4$, a set-associative random placement and random replacement (SA-RP+RR) cache with $S = 2$ and $W = 2$, and a fully-associative random replacement (FA-RR) cache with $W = 4$.

The number of cache layouts of those cache configurations is 15 for DM-RP, 26,244 for SA-RP+RR and 178,973,355 for FA-RR. It can be seen that the higher the associativity, the higher the number of cache layouts is. This is an expected result because the number of cache layouts for DM and FA caches depends on the number of unique cache line addresses (u) and the number of total cache accesses (m) respectively, and $u \leq m$. The number of cache layouts for SA caches must be always somewhere in the middle. This has an important implication in the worst-case performance of caches. Both caches have the same worst-case cache layout, i.e. the one in which all memory objects are mapped into the same cache entry, resulting in the largest number of cache misses. However, the probability of experiencing such cache layout is much lower for the FA-RR cache. In our example such probability is 1/15 and 1/178973355 for the DM-RP and FA-RR caches respectively. Hence, by using a set-associative cache with random placement and replacement policies (SA-RP+RR) the probability of experiencing the worst-case performance decreases rapidly with respect to the DM-RP: the number of cache layouts increases as the degree of randomness increases as well.

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 $\frac{1}{S}$ while the random replacement policy will evict a way to allocate a new fetched cache line with a probability of $\frac{1}{W}$. In particular, given the sequence $< A_i, B_1, ..., B_k, A_j >$, where A_i and A_j are two accesses to the same cache line and no B_l (where $1 \leq l \leq k$) accesses cache line A_i the probability of miss of A_j can be formulated as follows:

$$P_{miss_A j}(SA[S, W]) = P_{miss_A j}(DM[S]) \cdot P_{miss_A j}(FA[W]) \tag{6}$$

By combining Equation 1 and Equation 4 into Equation 6, the probability of miss of A_j can be approximated as:

$$P_{miss_{A_j}}(S,W) = \left(1 - \left(\frac{W-1}{W}\right)^{\sum_{l=1}^{l=k} P_{miss_{B_l}}}\right) \cdot \left(1 - \left(\frac{S-1}{S}\right)^{k}\right) \tag{7}$$

Such hit probability is used to compute the ETP of each cache access as follows where l_{hit} and l_{miss} are the cache hit and miss latency respectively:

$$\begin{aligned} ETP_{cache} = \{\{l_{hit}, l_{miss}\}, \\ \{P_{hit_{A_j}}(S,W), 1 - P_{hit_{A_j}}(S,W)\}\} \end{aligned} \tag{8}$$

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.

4 Hardware Design of a Random Cache

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

4.1 Random Replacement

Random replacement policies have been extensively used in various processor architectures, both in 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. Given that efficient implementations of a PRNG exist in the aforementioned processors, we omit the details of our implementation of the PRNG. The particular PRNG we have used in this paper is the Multiply-With-Carry (MWC) [17] 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.

4.2 Random Placement

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*, see Figure 2(b).

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, see Figure 2(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 cache placement. To be effective, the parametric hash function must have the following two properties:

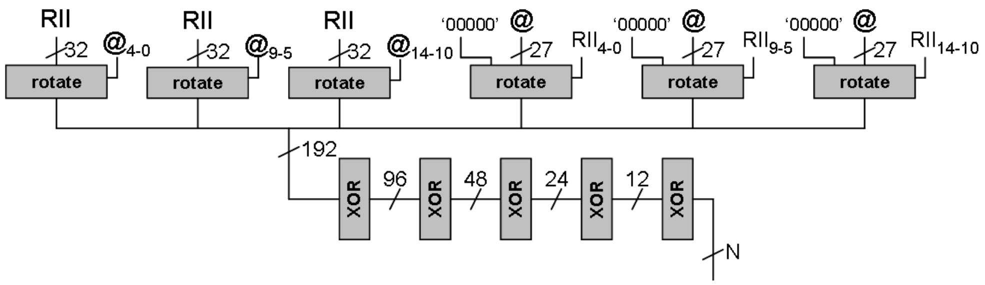

Fig. 3. Parametric hash function proposed for the random-placement cache.

- Small variations in the address bits or the RII must produce arbitrary variations in the output bits determining the set where the address is mapped.
- For any given pair of addresses, modifying the RII must produce different variations so that both addresses do not collide systematically in the same cache set. Ideally, both addresses should collide into the same set with a probability around $\frac{1}{S}$ for different RII values.

Those properties can be attained if the address and RII bits are used to control rotating blocks whose input is, for instance, the address to be accessed or the RII. In particular, RII bits must be used to control a rotate block[2] whose input is the address or vice versa. A single bit modification in either the address or the RII can change all output bits of the rotate block if such bit is one of the control bits. Note that using address bits to control a rotate block whose input is the address itself would be ineffective because changing the RII across runs would have no effect. Analogously using the RII bits only would also be ineffective because variations in the output of the rotate block would be identical for all addresses.

Figure 3 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 Figure 3, 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 bits) 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 three rightmost rotate blocks of the figure. By doing this, we ensure that when a different RII is used, the mapping of that address changes, and this change is different for different addresses. Analogously, the RII bits are rotated based on some address bits to obtain a different layout of RII bits for each address. This operation, which is performed by the three leftmost rotate blocks, changes the way the address bits 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.

2. Control bits of a rotate block are those determining how many positions the input bits are rotated.

Finally, all bits of the rotated addresses, the original address and the RII (192 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. The number of rotate blocks in each group is odd deliberately to avoid the case where bits controlling the rotation match across rotate blocks, thus producing the same output bits, which could systematically generate zeros when XORed.

As shown in Figure 3, the hardware implementation of the hash function consists of 6 rotate blocks and 5 levels of 2-input XOR gates. Each rotate block can be implemented with a 5-level multiplexer [12]. 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 hash function into the CACTI tool [18]. Results for several cache configurations show that energy per access grows less than 4% 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. Detailed power and delay results are provided later in Section 5.

5 Results

5.1 Experimental Setup

We use a cycle-accurate execution-driven simulator based on the SoCLib simulation framework [24], with PowerPC binaries [27]. 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); 2-way, 4-way, 8-way and 32-way (set-associative); and 256-way (fully-associative). Both caches implement random replacement and our random placement policy. We assume 100-cycles memory access latency.

The latency of the fetch stage depends on whether the access hits or misses in the instruction cache: 1

cycle in case of hit and 100 in case of miss. 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 model a single-core processor. WCET estimates are obtained for run-to-completion executions. Studying the interaction between our time-randomised cache design in the presence of system effects such as preemptions are part of our future work. Our initial results [13] show, that time-randomised caches simplify time composability, one of the most important metrics to optimise in current and future integrated real-time systems.

In order to evaluate our proposals, we use the EEMBC Autobench benchmark suite [19]. EEMBC Autobench reflects the current real-world demand of some automotive critical real-time embedded systems.

5.2 Quality of the Parametric Hash Function Implementation

We have evaluated the quality of our random placement function by using the test battery provided by the US National Institute of Standards and Technology [22]. Those tests evaluate the quality of the bit sequences produced by the PRNGs by studying the distribution of ones and zeros, their patterns, whether subpatterns repeat, etc. In particular, we have generated a sequence of 40,000,000 bits consisting of the set number produced by our parametric hash function given a particular random address and a sequence of 5,000,000 random numbers. Given that the cache considered is the same as above (4KB direct-mapped 16B/line), there are 256 cache sets and thus, each set identifier consists of 8 bits. Our hardware implementation of the parametric hash function passed 99.9% of the tests. None of the 9 PRNGs provided together with the test battery achieved a higher pass rate. Only two of them obtained the same pass rate, the *Secure Hash Generator* and the *Micali-Schnorr Generator* whose implementation is described in [22].

In the same experiment we have also tested the distribution across sets that our parametric hash function achieves. Given 5,000,000 set identifiers and 256 sets, we should expect our function to generate around 19,531 times each set identifier. Results show that the normalised standard deviation of the counts for the different sets is only 0.46% with respect to the expected value (90.4 with respect to 19,531). Maximum and minimum counts obtained are 19,769 and 19,333, so 1.2% and 1.0% away from the expected value.

Finally, we have tested how independent is the randomisation of different addresses. For that purpose we have generated 4 random addresses (say A, B, C and D) and have evaluated how many times addresses B, C and D are mapped into the same set as A for 2^{30} random seeds. In the ideal case we should obtain 2^{22} matches for each individual address (e.g., B being mapped into the same set as A), 2^{14} for each different pair of addresses (e.g., B and C being mapped into the same set as A simultaneously) and 2^{6} matches for the case where all addresses are mapped simultaneously in the same set. Results show that individual matches are within 0.2% of the ideal value (4,186,839 real vs. 4,194,304 expected), pairs are within 1.3% of the ideal value (16,591 real vs. 16,384 expected) and all of them collide in the same set 59 times when the expected value is 64 times.

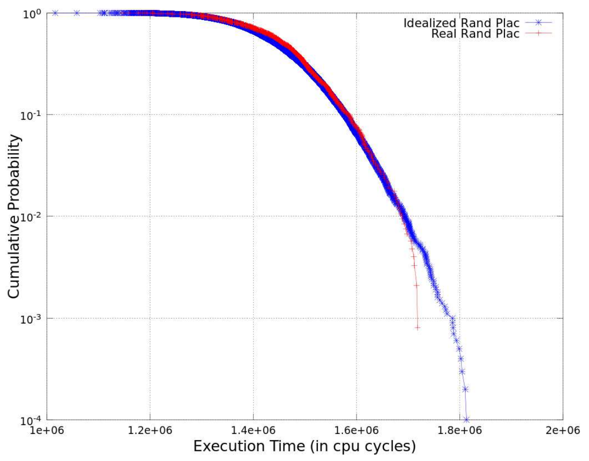

Fig. 4. 4KB direct-mapped cache considering an idealized random placement and the actual hardware implementation of the random placement (labelled as *Idealized Rand Plac* and *Real Rand Plac* respectively).

Overall, we can conclude that our hardware implementation of the parametric hash function achieves both (i) a high degree of randomisation, (ii) a near-optimal distribution of placement choices across sets and (iii) independent randomisation of addresses.

5.3 Behaviour of the Parametric Hash Function Implementation

In order to compare the behaviour of the implementation of our parametric hash function with an idealised random placement where each address is randomly mapped into one of the S sets, we compute the inverse cumulative distribution function (ICDF) of a representative benchmark, $a2time$ for both schemes (labeled *Real Rand Plac* and *Idealized Rand Plac* respectively) for a 4KB direct-mapped cache as shown in Figure 4. It can be seen that there is not meaningful difference between the two ICDF. Results only differ slightly in the tail of the queue. This indicates that the actual implementation of the parametric hash function randomises address mapping quite well across sets. Discrepancies in the tail of the queue can occur due to the loss of accuracy of the measurements for that region[3]. The results obtained for the other benchmarks match those presented in Figure 4.

3. Few measurements build the tail of the distribution. The particular behaviour of each of those measurements due to purely random events may create deviations with respect to the real ideal distribution.

TABLE 3
IPC of RP+RR and modulo+LRU caches in our architecture

		1w-256s DM	2w-128s SA	4w-64s SA	8w-32s SA	32w-8s SA	256w-1s FA
RP+RR	avg	2.10	1.78	1.73	1.73	1.74	1.75
	std	0.32	0.12	0.08	0.06	0.04	0.01
mod+LRU	avg	2.67	1.57	1.60	1.61	1.64	1.74

5.4 Fulfilling the i.i.d properties

The use of random replacement and placement policies guarantees that an ETP exists by construction for each memory operation, and so observed execution times fulfil the properties required by MBPTA. We further verify this point empirically by analysing whether execution times of EEMBC benchmarks on 6 different cache configurations (see Section 5.1) 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 are 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 [9]. 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.

Our results show that output values for the independence test values are largely below the threshold (1.96 for a 5% threshold) and p-values for the KS test are well above 0.05, thus proving that all our cache designs provide i.i.d. as needed by PTA.

5.5 Performance Analysis

Next, we compare the average performance of deterministic and random caches. In particular, we compare different *random placement+replacement* (RP+RR) caches against *modulo placement and LRU replacement* (mod+LRU) caches for different associativities. Table 3 shows the average CPI (cycles per instruction) for all EEMBC benchmarks under different cache configurations and 1,000 runs per benchmark for RP+RR caches. Standard deviation is also shown for RP+RR caches.

- As shown, execution time variation is quite stable for cache in the range 4-way to 256-way. Most of the benchmarks get little benefit due to the extra associativity and, in fact, some of them lose some performance (higher CPI) when increasing the associativity due to capacity misses. Note that in the extreme case, a program with a working set slightly larger than the cache size suffers more misses in a fully-associative cache than in a lowly-associative one. This is so because in lowly-associative caches some data may fit in few particular sets and deliver extra hits in front of a fully-associative one where all data are replaced short before being reused. This is the case for some programs in our setup (for instance, *aifftr* and *aiifft*). This effect is less noticeable for RP+RR caches because randomness breaks systematic pathological cases.
- When associativity becomes very low (2-way caches), most of the benchmarks observe little performance variation for mod+LRU caches and those exceeding slightly cache capacity get further improvements (so lower CPI). As stated before, this effect is not so relevant for RP+RR caches, where some benchmarks observe some execution time degradation with 2-way caches due to the conflicts introduced by random placement in some runs.
- Finally, if 1-way (direct-mapped) caches are used, conflicts dominate cache behaviour. While this introduces plenty of pathological cases in mod+LRU caches, pathological cases occur only randomly for RP+RR caches. Therefore, RP+RR caches offer lower execution time (so lower CPI) than mod+LRU under direct-mapped setups.

If we consider execution time variations in RP+RR caches, we observe that decreasing cache associativity may lead to more extreme scenarios. For instance, if a program accesses few different cache lines, those can evict each other some times in a fully-associative cache until they are finally placed in different cache lines. If we use a lowly-associative cache (e.g., 2-way or 4-way) most of the times those addresses will be placed into different cache sets, thus leading to slightly higher performance (so lower CPI) than a fully-associative cache. However, in some cases those addresses will be mapped into the same cache set and will experience many more conflicts, thus increasing the execution time (and so the CPI). As a consequence, although the average CPI is roughly the same for RP+RR cache configurations between 2-way and 256-way, their CPI variation increases as associativity decreases. As shown later, this variation has a direct impact on the pWCET estimates obtained for low exceedance probabilities. Just as a matter of fact, if we consider the average execution time plus 3 times the standard deviation ($\mu + 3 \cdot \sigma$), we can expect the execution time of around 99.9% of the runs to be below this value, so we should expect only 1 every 1,000 runs to exceed it. Then, we realise that this value is 1.78 for 256-way caches and 2.14 for 2-way caches. Thus, although higher associativity may not provide better average CPI, it provides more stable performance.

Example: Detailed results for *tblook*, *rspeed* and *aifftr* benchmarks are shown in Figures 5, 6 and 7 respectively. *tblook* shows some significant CPI degradation when associativity decreases down to 2-way and 1-way. Also, variability for RP+RR caches increases as associativity decreases. *rspeed* instead is the example of a highly stable benchmark fitting very well in cache so that its CPI is highly insensitive to cache associativity. Only for direct-mapped RP+RR caches

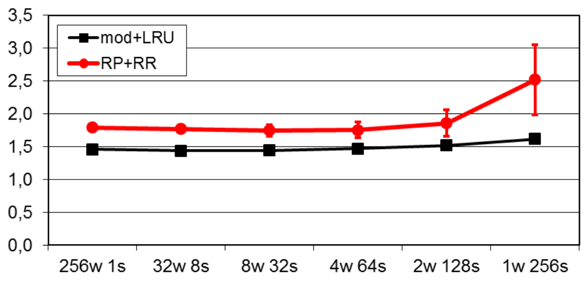

Fig. 5. CPI (cycles per instruction) for *tblook* benchmark for some RP+RR and LRU+mod cache configurations. In particular, we show, from left to right, FA, 32-way, 8-way, 4-way, 2-way and DM caches.

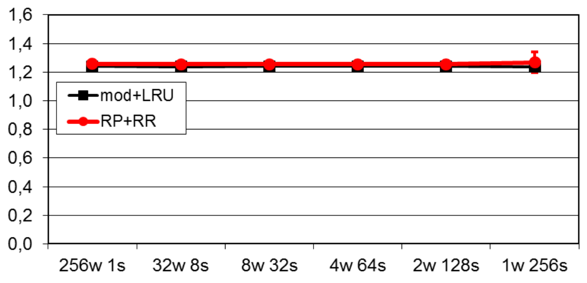

Fig. 6. CPI (cycles per instruction) for *rspeed* benchmark for some RP+RR and LRU+mod cache configurations. In particular, we show, from left to right, FA, 32-way, 8-way, 4-way, 2-way and DM caches.

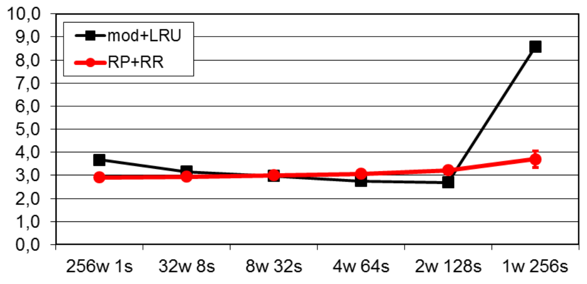

Fig. 7. CPI (cycles per instruction) for *aifftr* benchmark for some RP+RR and LRU+mod cache configurations. In particular, we show, from left to right, FA, 32-way, 8-way, 4-way, 2-way and DM caches.

variability increases due to few executions where those few cache lines reused are randomly mapped into the same cache set. *aifftr* is a clear example of a program not fitting completely in cache. Therefore, its execution time decreases as associativity decreases for mod+LRU configurations, except when a direct-mapped cache is used because then conflicts dominate its behaviour. Instead, RP+RR caches show much more stable performance across different cache setups. Conflicts across different cache lines are abundant in all setups, but extreme conflicts never occur due to the low probability of extreme scenarios. This can be easily explained with an example. If we flip 1,000 coins, it is extremely unlikely to get a number of tails (or faces) out of the range 400-600. In the case of mod+LRU, those 1,000 coins have been glued to each other, so whatever it happens, is either very good (no conflict at all) or very bad (systematic conflicts).

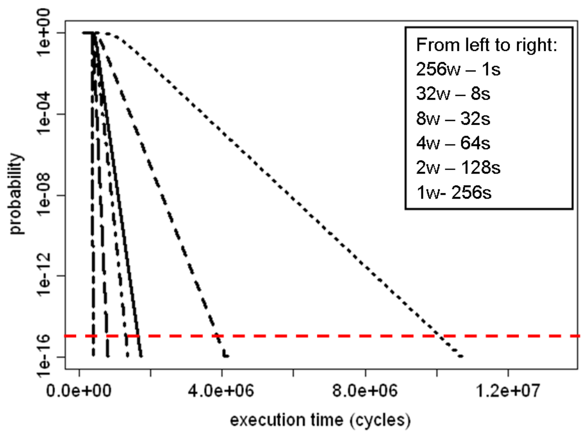

Fig. 8. EVT projection for *a2time*.

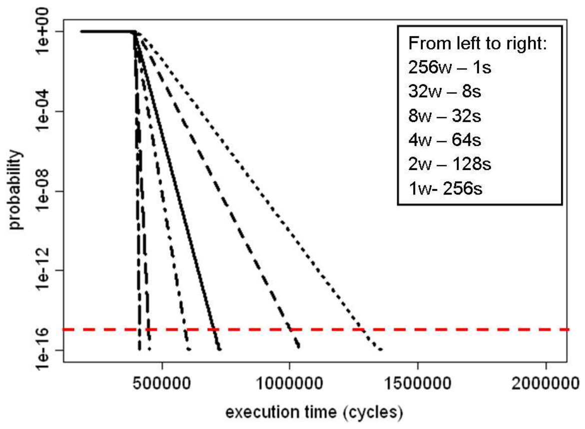

Fig. 9. EVT projection for *ttsprk*.

5.6 MBPTA: EVT projections

In this section we provide several pWCET estimates obtained with the method provided in [9]. Note that MBPTA has been used so far *only* on top of FA-RR caches. Although FA-RR caches fulfil 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.

Our selection of the exceedance probability, 10^{-15}, 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 (so 180,000 times per hour) [1]. Hence, the highest allowed failure rate per task activation is 5.6×10^{-15}, which is largely above our exceedance probability.

Following the iterative method in [9] we carried out 1,000 experiments and used EVT to extract pWCET estimates. Figures 8 and 9 show the EVT projections generated with MBPTA [9] for `a2time` and `ttsprk` considering a FA-RR cache (labelled as *256w-1s*), several set-associative RP+RR caches (labelled as *32w-8s*, *8w-32s*, *4w-64s* and *2w-128s*) and a DM-RP cache

TABLE 4
pWCET increment of the SA and DM caches with respect to the FA one, considering an exceedance probability of 10^{-15}

Benchmarks	32w-8s (SA)	8w-32s (SA)	4w-64s (SA)	2w-128s (SA)	1w-256s (DM)
a2time	95%	228%	311%	862%	2404%
aifftr	39%	41%	68%	138%	532%
aifirf	111%	152%	317%	452%	777%
aiifft	38%	53%	54%	97%	571%
cacheb	1%	10%	16%	75%	1008%
canrdr	11%	26%	90%	191%	614%
iirflt	178%	329%	419%	543%	1945%
puwmod	16%	36%	45%	250%	626%
rspeed	9%	22%	84%	230%	475%
tblook	56%	152%	185%	299%	784%
ttsprk	9%	43%	70%	142%	210%

(labelled as *1w-256s*). 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 as shown in Figure 8. For an exceedance probability of 10^{-15}, the pWCET of a 8-way cache for *a2time* grows 311% with respect to the FA-RR cache and more than 24x in the case of the DM-RP cache. Instead, *ttsprk* experiences pWCET estimate increments of only 9% and 43% when considering 32-way and 8-way caches and around 3x when considering a DM-RP cache, with respect to the FA-RR cache.

Table 4 shows the pWCET increment of all set-associative and direct-mapped caches with respect to the fully-associative one for all benchmarks when considering an exceedance probability of 10^{-15}. 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.

The lower the associativity the higher the pWCET bound is. The reason for that behaviour is that, although average performance does not change noticeably across cache configurations, execution time variation grows when decreasing cache associativity. While this variation may be relatively small in the 1,000 runs of a particular benchmark for a given cache setup, it may grow orders of magnitude at very low probabilities (e.g., 10^{-15}), and so MBPTA must account for that. Thus, pWCET estimates for direct-mapped caches are around 10X those for fully-associative ones even if average execution time is only 20% higher. For reasonable cache setups pWCET increments with respect to the ideal fully-associative cache are far more moderate. For instance, they are 99% and 151% higher for 8-way and 4-way caches respectively than for fully-associative caches. Note that, as stated earlier, cache sizes (4KB) have been chosen to create conflict and capacity misses in several benchmarks. Thus, if larger caches are used instead (e.g., 8KB or 16KB) or if L2 caches are set up to mitigate miss latencies, then pWCET estimates for lowly-associative caches would get closer to those of fully-associative ones.

Results have been obtained assuming that cache latency is not increased due to the hash function. If this is not the case, cache latency should be increased by up to 1 cycle. We have corroborated that, as stated before, increased cache latency has negligible impact in pWCET estimates, which grow on average between 1.1% and 5.5% only across different associativities.

5.7 Power and Delay Analysis

This section evaluates the power and delay overhead of the parametric hash function when used together with different cache configurations. We have integrated our parametric hash function into the CACTI tool [18] and have run experiments for a set of configurations. Cache sizes considered are 4KB, 8KB, 16KB, 32KB and 64KB; associativities are 1-way (direct-mapped), 2-way, 4-way, 8-way and fully-associative. Cache line size has no meaningful impact, so we have considered 16 bytes per line as in the rest of the paper. Other relevant parameters are technology node (32nm), type of transistors (low operating power ones), access mode (sequential tag and data access for low power) and ports (1 read/write port).

Table 5 shows the relative energy overhead per read access for all non-FA configurations implementing RP with respect to the design without the parametric hash function, whose energy consumption ranges between 3.8pJ and 22.2pJ per access for different configurations. Also, the overhead of a FA cache implementing random replacement is shown with respect to the same baseline (its energy consumption ranges between 32.9pJ and 474.3pJ per access for different configurations). We observe that the energy overhead per read access ranges between 1.5% and 8.3% across all configurations for our RP cache (around 0.3pJ). Since such parametric function is independent of the cache size, the larger the cache size, the lower the relative energy overhead is. This overhead is largely below that incurred by a PTA-friendly FA cache whose energy overhead is between 588% and 2107%.

Analogously to energy, Table 6 shows the relative access time increase per read access. The relative access time increase for our RP cache is between 28% and 54% (around 0.2ns), being lower for larger caches since the delay of the parametric hash function is almost independent of the cache size, whereas delay for caches increases from around 0.37ns (4KB) to around 0.71ns (64KB). In fact, for sufficiently large

TABLE 5
Relative energy increase of (i) random placement (top rows) and (ii) fully-associative (bottom rows) caches w.r.t. modulo placement cache designs

	8-way RP vs base	**4-way RP vs base**	**2-way RP vs base**	**DM RP vs base**
4KB	6.8%	7.7%	8.3%	8.3%
8KB	4.8%	5.4%	5.6%	5.7%
16KB	3.4%	3.5%	3.6%	3.6%
32KB	2.2%	2.3%	2.3%	2.4%
64KB	1.5%	1.5%	1.5%	1.5%
	FA vs 8-way base	**FA vs 4-way base**	**FA vs 2-way base**	**FA vs DM base**
4KB	588.3%	680.3%	743.2%	743.6%
8KB	822.8%	932.1%	959.5%	981.9%
16KB	1199.0%	1220.7%	1254.8%	1272.2%
32KB	1559.4%	1608.6%	1630.2%	1662.3%
64KB	2039.0%	2074.4%	2107.0%	2085.3%

TABLE 6
Relative access time increase of (i) random placement (top rows) and (ii) fully-associative (bottom rows) caches w.r.t. modulo placement cache designs

	8-way RP vs base	**4-way RP vs base**	**2-way RP vs base**	**DM RP vs base**
4KB	54.4%	52.7%	49.1%	52.6%
8KB	49.3%	45.8%	45.6%	45.6%
16KB	39.2%	40.5%	40.5%	40.5%
32KB	34.6%	34.6%	34.6%	34.7%
64KB	28.3%	28.3%	28.4%	29.2%
	FA vs 8-way base	**FA vs 4-way base**	**FA vs 2-way base**	**FA vs DM base**
4KB	60.3%	55.5%	44.8%	55.1%
8KB	275.6%	248.6%	247.1%	247.1%
16KB	840.8%	872.8%	872.8%	872.8%
32KB	2918.9%	2919.2%	2919.3%	2929.7%
64KB	9365.1%	9367.3%	9396.9%	9664.9%

caches fewer XOR levels may be needed, thus further reducing the relative delay of the RP caches. Instead, FA caches have much higher access times for 8KB caches and above (between 1.53ns and 67.27ns), and a bit higher average access time for tiny 4KB caches (0.59ns). Overall, the access time may only impact hit latency, which, as stated before, is not the dominant factor in WCET.

Overall, our RP cache is a much more efficient design than exotic fully-associative caches enabling PTA. Furthermore, the larger the cache size, the lower the relative overheads introduced by our parametric hash function. Thus, our design is both efficient and scalable.

6 Related Work

WCET impact of caches has been studied extensively [21], including several levels of cache [15] and locking mechanisms [20] to increase predictability and hence, provide tighter WCET estimates.

Some current processors used in high-performance embedded systems already implement random replacement policies 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 [25][23]. 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 [10]. 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 [7][9]. 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 [16]. 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 [7][9].

7 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 attach 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 obtained, 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 TLB.

Acknowledgments

This work has been mainly supported by the PROARTIS FP7 European Project under grant agreement number 249100. It has also been partially supported by the Spanish Ministry of Science and Innovation under grant TIN2012-34557 and HiPEAC. Leonidas Kosmidis is funded by the Spanish Ministry of Education under the FPU grant AP2010-4208. Eduardo Quiñones is partially 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(F) 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] J.M. Cargal. *Discrete Mathematics for Neophytes: Number Theory, Probability, Algorithms, and Other Stuff*. 1988.
[7] 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. *ACM Transactions on Embeddec Computing Systems (TECS)*, 2013.
[8] R.N. Charette. This car runs on code. In *IEEE Spectrum online*, 2009.
[9] 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 *Proceedings of the 24th Euromicro Conference on Real-Time Systems*, ECRTS '12, 2012.
[10] L. David and I. Puaut. Static determination of probabilistic execution times. In *ECRTS*, 2004.
[11] Ronald L. Graham, Donald E. Knuth, and Oren Patashnik. *Concrete Mathematics*. Addison-Wesley, Reading MA., 1988.
[12] S. Huntzicker, M. Dayringer, J. Soprano, A. Weerasinghe, D.M. Harris, and D. Patil. Energy-delay tradeoffs in 32-bit static shifter designs. In *ICCD*, 2008.
[13] L. Kosmidis, E. Quinones, J. Abella, T. Vardanega, and F.J. Cazorla. Achieving timing composability with probabilistic timing analysis. In *ISORC*, 2013.
[14] Samuel Kotz and Saralees Nadarajah. *Extreme value distributions: theory and applications*. World Scientific, 2000.
[15] B. Lesage, D. Hardy, and I. Puaut. WCET analysis of multi-level set-associative data caches. *WCET Workshop*, 2009.
[16] Y. Liang and T. Mitra. Cache modeling in probabilistic execution time analysis. In *DAC*, 2008.
[17] G. Marsaglia and A. Zaman. A new class of random number generators. *Annals of Applied Probability*, 1(3):462–480, 1991.
[18] N. Muralimanohar, R. Balasubramonian, and N.P. Jouppi. CACTI 6.0: A tool to understand large caches. *HP Tech Report HPL-2009-85*, 2009.
[19] Jason Poovey. *Characterization of the EEMBC Benchmark Suite*. North Carolina State University, 2007.
[20] I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS*, 2002.
[21] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37:99–122, November 2007.
[22] A. Rukhin, J. Soto, J. Nechvatal, M. Smid, E. Barker, S. Leigh, M. Levenson, M. Vangel, D. Banks, A. Heckert, J. Dray, and S. Vo. A statistical test suite for the validation of random number generators and pseudo random number generators for cryptographic applications. Special publication 800-22rev1a, US National Institute of Standards and Technology, 2010.
[23] A. Seznec and F. Bodin. Skewed-associative caches. In *PARLE*. 1993.
[24] SoCLib. -, 2003-2012. http://www.soclib.fr/trac/dev.
[25] Nigel Topham and Antonio González. Randomized cache placement for eliminating conflicts. *IEEE Transactions on Computers*, 48:185–192, February 1999.
[26] F. Wartel, L. Kosmidis, C. Lo, B. Triquet, E. Quinones, J. Abella, A. Gogonel, A. Baldovin, E. Mezzetti, L. Cucu, T. Vardanega, and F.J. Cazorla. Measurement-based probabilistic timing analysis: Lessons from an integrated-modular avionics case study. In *SIES*, 2013.
[27] J. Wetzel, E. Silha, C. May, B. Frey, J. Furukawa, and G. Frazier. *PowerPC User Instruction Set Architecture*. IBM Corporation, 2005.
[28] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G.Bernat, C. Ferdinand, R.Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, G. Staschulat, and P. Stenströem. The worst-case execution time problem: overview of methods and survey of tools. *ACM TECS*, 7(3):1–53, 2008.

Leonidas Kosmidis is a PhD. student in the CAOS group at Barcelona Supercomputing Center (BSC), Spain. Leonidas joined BSC in 2009 and received his MSc in Computer Architecture, Networks and Systems in 2011 from Universitat Politècnica de Catalunya (UPC), Spain. He holds a BSc in Computer Science from University of Crete, Greece and worked as an intern at ARM Cambridge and École Centrale Paris. Leonidas participates in FP7 PROARTIS and PROXIMA projects. His main focus is hardware and software design for real-time embedded systems.

Jaume Abella is a senior PhD. Researcher in the CAOS group at BSC and member of HIPEAC. He received his MS (2002) and PhD. (2005) degrees from the UPC. He worked at the Intel Barcelona Research Center (2005-2009) in the design and modelling of circuits and microarchitectures for fault-tolerance and low power, and memory hierarchies. He joined the BSC in 2009 where he is in charge of hardware designs for FP7 PROARTIS and PROXIMA, and BSC tasks in ARTEMIS VeTeSS. Jaume is also involved in two ESA-BSC bilateral projects and FP7 parMERASA. He has authored more than 15 patents and 60 papers in top conferences and journals. He is (has been) co-advisor of ten MS and PhD students.

Eduardo Quiñones is a senior PhD. Researcher at BSC and member of HiPEAC. He received his MS degree in 2003 and his PhD. in 2008 at the UPC. His area of expertise is in safety-critical systems and high performance compiler techniques. He is involved in several FP7 European projects (parMERASA, PROARTIS, PROXIMA and P-SOCRATES), as well as some bilateral ESA-BSC projects. He spent one year as a student intern at Intel Research Labs (2002 - 2003).

Francisco J. Cazorla is the leader of the CAOS group at BSC and member of HIPEAC Network of Excellence. He has led projects funded by industry (IBM and Sun Microsystems), by the European Space Agency (ESA) and public-funded projects (FP7 PROARTIS project and FP7 PROXIMA project). He has participated in FP6 (SARC) and FP7 Projects (MERASA, VeTeSS, parMERASA). His research area focuses on multithreaded for both high-performance and real-time systems on which he is co-advising several PhD theses. He has co-authored 3 patents and over 70 papers in international refereed conferences and journals.