Locality-aware Cache Random Replacement Policies

Pedro Benedicte[a,b,*], Carles Hernandez[a,d], Jaume Abella[a], Francisco J. Cazorla[a,c]

Carrer Jordi Girona 29, Barcelona, Spain

[a] *Barcelona Supercomputing Center (BSC), Barcelona, Spain*
[b] *Universitat Politècnica de Catalunya (UPC), Barcelona, Spain*
[c] *IIIA-CSIC, Bellaterra, Spain*
[d] *Universitat Politècnica de València (UPV), València, Spain*

Abstract

Measurement-Based Probabilistic Timing Analysis (MBPTA) facilitates the analysis of complex software running on hardware comprising high-performance features. MBPTA also aims at preventing additional analysis costs for timing analysis techniques and preserving the confidence on derived WCET estimates. Cache behavior has a deep influence on WCET estimates and hence on "the amount of software" that can be consolidated onto a single hardware platform. Deterministic replacement policies such as LRU (Least Recently Used) and NMRU (Non-Most Recently Used) have systematic pathological cases that may lead to high execution times and WCET estimates. Instead, random replacement (RR) decreases pathological cases probability, at the cost of temporal locality.

We present two new MBPTA-amenable replacement policies that completely remove the presented pathological cases. The first policy, Random Permutations (RP) preserves higher temporal locality than RR; while the second, NMRU Random Permutations (NMRURP), also protects the Most Recently Used line from eviction. Both proposed policies build upon restricted random replacement choices. Our simulation evaluation (validated against a real prototype) using the Malardalen benchmarks and a case study shows that RP and NMRURP deliver both high average performance (within 1% of LRUs and NRMU performance) and tight WCET estimates 11% and 24% lower than those of RR.

1. Introduction

[1] The increased level of automation in critical real-time embedded systems (CRTES) and the advent of autonomous vehicles across different CRTES domains call for higher levels of performance than those provided by simple microcontrollers used nowadays [2, 3]. Such performance can only be achieved using powerful processors implementing high-performance features including cache memories. Unfortunately, caches challenge the estimation of Worst-Case Execution Time (WCET), a fundamental step for timing validation and verification and hence for assessing the correct timing behavior of CRTES [4].

In this work, we build upon measurement-based probabilistic timing analysis (MBPTA) [5, 6]. MBPTA is a mature technology that has been successfully assessed with case studies in the automotive, railway, space, and avionics domains [7][8]. MBPTA builds on the underlying (complex) platform having certain properties in its timing behavior as a means to facilitate the analysis of software timing. In particular, MBPTA requires the sources of execution time variability (jitter) to be either upper-bounded (during analysis) or time-randomized (during both analysis and operation). For instance, latencies due to values operated in variable-latency units are typically upper-bounded, and cache placement is typically randomized. By applying these techniques to the different sources of jitter of the processor, MBPTA relieves the user from controlling during the test campaign low level aspects of those resources causing jitter, for which the end user may lack means to determine and en-

*Corresponding author
Email address: pbenedic@bsc.es (Pedro Benedicte)

[1] This work is based on the work previously published in SAC [1]

force their worst timing behavior at analysis.

The jitter that caches cause on programs' execution time challenges the tightness of the bounds derived by timing analysis techniques since memory placement (and so cache placement) changes across software integrations [9]. If these effects are not properly leveraged – which is in general quite challenging – WCET estimates obtained for a given software unit are no longer valid when incrementally integrated with other software units (even if they share no data or instructions) as part of the regular software development process. This occurs because small changes in the memory placement modify cache layouts and hence, cache performance [9]. With random placement, which varies across runs, and replacement caches [10][11] the space of potential cache mappings is naturally and randomly explored as the number of program runs (tests) performed is increased. By preserving random placement and replacement during operation, the timing behavior of caches matches probabilistically that explored at analysis, effectively relieving the end user from controlling whether bad cache mapping scenarios are captured during experiments carried out at analysis.

MBPTA's requirements on cache memories can be achieved either with software or hardware means. Software solutions have been implemented as a pass of the LLVM compiler [10] and as a source-to-source compiler [12], with different implications on the interpretation of WCET estimates. Hardware solutions (the focus of this work), while requiring specific implementations provide in general higher performance. Cache hardware randomization techniques have been progressively assessed in performance simulators [13] and higher-maturity solutions at RTL level on LEON-based platforms (widely deployed in the space domain) that are now offered as a commercial product [14].

The impact of caches on execution time is highly program-dependent. For instance, some programs barely exploit cache space. In the context of CRTES, programs may be produced by means of automatic code generation tools (e.g. SCADE [15]), typically leading to programs with few thousands of instructions. For these programs, random replacement (RR) may produce a first pathological scenario ($ps1$) in which few cache lines fitting in a cache set, randomly evict each other on each miss despite there are some available lines in that set. This occurs because nothing prevents unfortunate replacement choices whose probability decreases, but still must be accounted for with WCET estimates, which may be relatively high. RR can also cause a second pathological scenario ($ps2$) resulting in the replacement of cache lines that have been just accessed, losing some temporal locality. Both effects have the same root cause: cache lines recently fetched can be randomly evicted shortly after fetched. In order to tackle this challenge, we make the following contributions:

1. We make an in-depth analysis of pathological behavior of RR in terms of probability of each pathological scenario and its potential impact on performance.
2. We propose *Random Permutations* (RP) and Non-Most Recently Used Random Permutation (NMRURP) that restrict random choices to prevent pathological cases. RP evicts *all* lines in a set, yet in a random order, before it starts evicting occupied lines, whereas NMRURP additionally prevents the last cache line accessed from being replaced. Both techniques reduce the number of evictions that can occur before all lines are placed in a cache set ($ps1$) and increase temporal locality ($ps2$) by reducing the chance of evicting recently accessed lines.
3. We perform a detailed analysis of the suitability of the presented replacement policies, both deterministic policies and their random counter-parts, with respect to the properties needed by MBPTA.
4. Finally, we assess RP and NMRURP complexity and benefits with the Mälardalen benchmarks as well as a railway case study, providing evidence of their feasibility and gains in terms of WCET reduction.

The rest of the paper is organized as follows. Sections 2 and 3 respectively introduce MBPTA and existing replacement policies. RP and NMRURP are presented in Section 4. Section 5 reviews the MBPTA compliance of the different replacement policies. RP and NMRURP are evaluated in Section 6. Section 7 describes some related work. Finally, conclusions are drawn in Section 8.

2. Background on MBPTA

Safety standards require deriving WCET estimates for software units as basic building block for assessing the overall timing of the software.

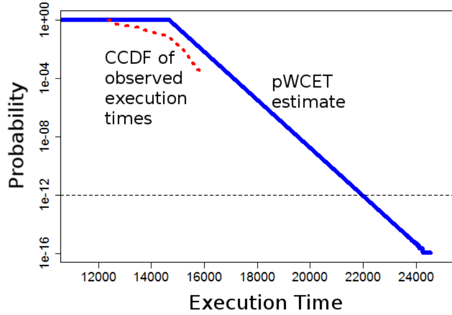


Figure 1: MBPTA application example to an execution time sample with 1,000 measurements. The red dashed line is to the empirical CCDF of the execution time sample. The blue solid line is the EVT-derived pWCET distribution.

WCET estimates need to be as tight as possible and be consistent with safety-related requirements (goals). A common misconception is that a WCET estimate overrun necessarily causes a system level failure. This, however, is not true since safety mechanisms factor in the impact that overruns can have on the safety goals. Following the probabilistic approach used to handle random hardware faults [16][17], MBPTA reasons on WCET as a distribution, aka probabilistic WCET (pWCET) curve [18, 19], describing the maximum probability with which a WCET estimate can be exceeded. This approach has been assessed in case studies in railway, space, aerospace and automotive [7][8]; and shown fit safety standards [20]. We refer the reader to those works for more details on MBPTA.

MBPTA builds on a set of execution time measurements taken during system analysis phase, whose Complementary Cumulative Distribution Function (CCDF) is shown with a dashed red line in Figure 1. Those measurements are passed as input to Extreme Value Theory (EVT) [21], a statistical tool to estimate an upper-bound distribution for high execution times. MBPTA controls how execution time measurements are collected so they capture those conditions that lead to higher or equal execution times than those during system operation. EVT requires that the execution times meet several statistical properties related to the degree of independence and identical distribution of the random variable (execution times) modeled [22, 23]. Also whether execution times can be modeled with an exponential tail, which is the most convenient distribution for pWCET estimates of real-time programs [5, 6].

MBPTA delivers a pWCET distribution, often depicted as a CCDF, so that for each particular execution time value we obtain an upper-bound probability with which it can be exceeded (see blue solid line in Figure 1). Therefore, the pWCET estimate is the value such that its upper-bound exceedance probability can be regarded as irrelevant in relation to acceptable failure rates in the corresponding safety standards (e.g. ISO26262 in automotive [16]). For instance, as shown in Figure 1, the probability of the program to take 22,000 cycles or longer is below an exceedance probability of 10^{-12} per run.

Interestingly, EVT is able to predict the probabilities for combinations of events that have not occurred simultaneously in any of the observations in the sample. For instance, if those observations correspond to the execution time when experiencing between 10 and 20 cache misses, EVT can predict the probabilities for execution times caused by a larger number of misses (e.g. caused by 50, 100 or 1,000 misses) [24].

3. Analysis of replacement policies

Several replacement policies have been proposed over the years. We classify them into two main categories: those with a fully deterministic behavior and those with a randomized behavior.

3.1. Deterministic replacement policies

We analyze LRU, NMRU and BT replacement policies as a representative of deterministic policies. It is noted that the latter two have been proposed to reduce the hardware requirements of the former [25]. Our analysis shows that all of them have systematic pathological cases, i.e. addresses sequences for which they result in evictions that occur systematically.

LRU keeps the order in which lines in the set have been last accessed, from the most recently used (MRU) to the least recently used (LRU). Upon a cache hit, the cache line accessed is promoted to the top of the list (MRU). Upon a cache miss, the cache line in the botton of the list (LRU) is replaced and used to store the newly fetched cache line, which is then promoted to the first position in the list.

NMRU can be seen as a simplified version of LRU. Keeping the full order of cache lines in a set is increasingly costly for high associative cache memories: for an N-way cache, LRU requires keeping the

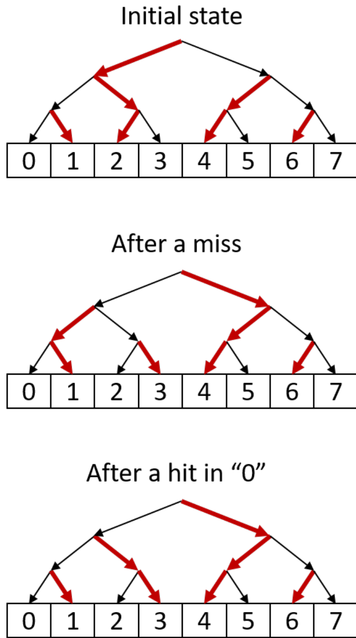


Figure 2: Schematic of BT operation. Dark thick lines indicate candidates for replacement.

full order across the N elements. NMRU instead only prevents the MRU line from being evicted, without imposing any particular order on the other cache lines. This can be achieved, for instance, with a single pointer indicating the next line to be evicted in the set (e.g. in a round-robin fashion) skipping the MRU line if it is selected for eviction. In general, such a solution may not preserve temporal locality as much as LRU, since the 2nd MRU line could be evicted instead of the LRU line on a miss. On the positive end, NMRU scales to arbitrarily highly associative caches with limited cost: a pointer to the next line to be evicted (incremented upon an eviction) and another pointer to the MRU line to protect it from being evicted.

BT protects the MRU line, as NMRU does, but also keeps some partial order on the remaining lines. Hence, BT lays in-between LRU and NMRU in terms of ability to preserve temporal locality and complexity. In particular, BT partitions cache lines in a set into two halves (left and right) recursively indicating in each partition the side from where to select the line to be evicted. This is illustrated in Figure 2 for an 8-way cache set. In the figure, cache line 2 is the candidate to be replaced. Upon an access, all arrows in the path to the cache line accessed are changed to point to the other element. For instance, as shown in the figure, on a miss all arrows pointing to 2 are changed, thus pointing now to 4. Then, upon a hit on cache line 0, only the arrow pointing to the pair 0-1 is changed to point to the pair 2-3. Note that within the pair 2-3, the corresponding arrow points to 3, thus providing some temporal locality protection for 2, which has been accessed recently. While such protection does not guarantee full order as in the case of LRU, it provides some further temporal locality protection than in the case of NMRU. BT has a logarithmic cost on the associativity, thus limiting the overhead w.r.t. LRU, although it is slightly higher than that of NMRU. In particular, it requires 1 bit for each left/right choice. Hence, N-1 bits are needed per set for an N-way cache.

Pathological cases. All deterministic policies have, at least, one pathological case for which accesses systematically result in misses. One such cases for all deterministic policies above is an access sequence with N+1 different addresses accessed in a round-robin fashion.

For instance, for a 4-way cache, a pathological case would be repeating the sequence $ABCDE$ an arbitrary number of times.

- LRU: after the first 4 accesses, A is the LRU line, so E evicts A. Then we access A, which is a miss and evicts B. Then access B results in a miss that evicts C, and so on and so forth.
- For NMRU it can be seen that 5 different addresses are enough to evict cache lines in the 4 sets in a round-robin fashion. In this particular example we never try to evict the MRU line, so the behavior is analogous to a FIFO policy.
- In the case of BT, although less obvious, after 4 misses, the tree points to the first line fetched out of those for replacement systematically, so E evicts A, A evicts B, B evicts C and so on and so forth, analogously to the case of LRU and NMRU.

While our analysis is limited to a subset of replacement policies, it serves the purpose of illustrating that systematic pathological cases exist for deterministic policies due to their intrinsic deterministic nature. Such systematic cases can only be broken, in general, by means of some form of random choice. For instance, this is the case of random replacement (RR), which randomly selects the cache line to be evicted in the set upon a miss.

3.2. Random replacement

Cache memories achieving MBPTA compliance via hardware implement random placement and replacement [10, 11]. Random placement [10] produces random and independent address mappings across sets so that two arbitrary addresses (not in the same cache line) are mapped to the same set with a probability $1/S$, where S is the number of sets. Random replacement (RR), makes random eviction choices so that, in the event of a miss in a given set, for a cache with W ways, the probability of a line in that set to be evicted is $1/W$. RR builds on a pseudo-random number generator (PRNG) with sufficient quality to allow cache conflicts to be truly random. These low-cost and MBPTA-amenable probabilistic properties of such PRNGs have already shown elsewhere [26].

RR evicts a specific cache line with a probability of 1/N for an N-way cache. Hence, the probability of a line surviving an eviction is (N-1)/N, and hence, there is a non-null probability of survival for cache lines for any access sequence. In particular, for the systematic pathological case above for deterministic replacement policies, RR provides a survival (hit) probability of $\left(\frac{3}{4}\right)^4 = 0.316$.

On the other hand, if we consider a sequence with N addresses instead of N+1, then we realize that all deterministic policies would lead to all-hit sequences except for cold misses. Instead RR has non-null probability of evicting some cache lines before (randomly) placing all addresses in distinct physical cache lines so that all remaining accesses become hits. This occurs because RR is unaware of temporal locality since it does not preserve any history.

Since pWCET estimates with MBPTA need to account for the worst case that can occur with non-negligible probability, the pathological cases of deterministic policies need to be accounted. In the case of RR, some low-probability high-execution time eviction scenarios need to be accounted for, but those scenarios will not include the absolute worst case, which is the actual case for deterministic policies under systematic pathological cases. Still, the fact that these eviction sequences can occur under RR may lead to high pWCET estimates, which relates to the fact that RR is a locality-unaware replacement policy. This is illustrated with the following examples.

3.2.1. The number of objects mapped to a cache line is smaller than or equal to W

Let us consider a fully-associative data cache with 4 ways ($W = 4$) and a program accessing alternatively addresses A and B, which belong to different cache lines, 20 times each. First, A is placed in a random line. Then, B will be placed in a random line, with a probability of $3/4$ of not replacing A and $1/4$ of replacing it. If B replaces A, then A has a $1/4$ probability of replacing B again. And they can keep replacing each other with probability $1/4$. Overall, this program experiences $M + 2$ misses (2 cold misses are always experienced), where $M \geq 0$, with a probability:

$$P(M) = \left(\frac{1}{4}\right)^M \times \frac{3}{4} \tag{1}$$

For instance, $P(4)$ (the probability of having $4 + 2 = 6$ misses) is ~ 0.003, $P(10) \approx 7 \cdot 10^{-7}$ and $P(20) \approx 7 \cdot 10^{-13}$ (see blue line in the top chart of Figure 3). Assuming 10 cycles per miss and 1 cycle per hit, and considering only the impact of cache in execution time, the pWCET at an exceedance threshold of 10^{-12} per run can only be at least 238 cycles to account for 22 misses and 18 hits, since the probability of having at least 22 misses (the accumulated probability of [22,40] misses) is $\sim 9 \cdot 10^{-13}$, see blue line in the bottom chart of Figure 3.

3.2.2. The number of objects mapped to a cache line exceeds W

Another pathological case arises when the number of objects mapped to the same set exceeds W. In this case, RR can lead to the loss of temporal locality, since random eviction patterns can make recently touched objects (i.e. cache lines) be evicted from cache on a miss, whereas old-standing (unlikely to be reused) objects remain in cache. For instance, in the repeating sequence $ABACADAE...$, where A is continuously interleaved with accesses to addresses that lead to a cold miss, RR may evict A sometimes. Conversely, deterministic policies presented in previous section would always protect A from eviction since it is the MRU line upon the access to any other address. As the number of objects mapped per set increases, cold and capacity (unavoidable[2]) misses become the main contribu-

[2]They could only be avoided prefetching cache lines, but still prefetch requests would need to fetch data from upper memory levels.

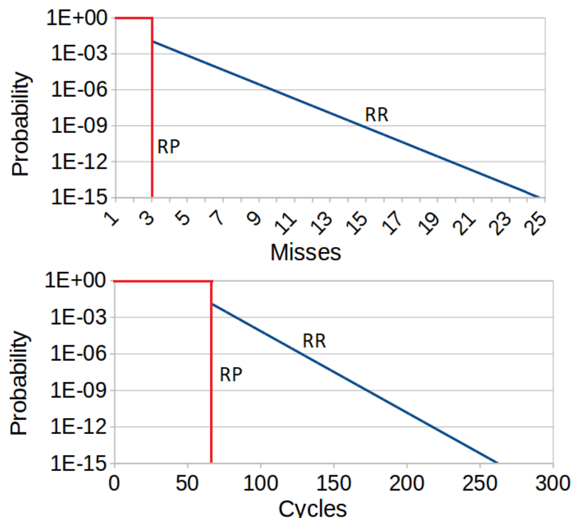


Figure 3: pWCET and probabilistic miss count curves.

tors to miss rates, naturally reducing the benefit of any replacement policy.

4. Locality-aware random replacement

In this section we introduce our proposed (temporal) locality-aware random replacement policies. In particular, we propose Random Permutations (RP) and NMRU Random Permutations (NMRURP), which aim at avoiding systematic pathological cases such as those of deterministic policies as well as preserving higher levels of temporal locality than RR, thus improving RR.

4.1. Random Permutations (RP)

RP limits pathological random replacement scenarios by increasing temporal reuse and enforcing random evictions to occur across all cache ways.

- When accessed data fits in a cache set, they will eventually be placed in different cache lines, thus avoiding potentially long mutual evictions by construction. This would result in a single replacement for the previous example, thus leading to a maximum execution time of 67 cycles (3 misses and 37 hits), see red lines in Figure 3.
- When the number of accessed lines exceeds the size of a set, RP effects are also positive increasing reuse, though the impact of replacement naturally reduces.

To reach its goals, RP leverages the concept of Random Permutations [27], which we first introduce and then explain how can be used and implemented in RP. We finally show how the resulting RP limits pathological eviction patterns.

4.1.1. Logic behind Random Permutations

Random Permutations avoid potentially infinite starvation in the arbitration for shared resources, where one requester (unluckily) loses all arbitrations with decreasing probabilities. This occurs with standard random (lottery) arbitration [28], with which on every arbitration round the grant is randomly given to one of the requesters without taking into account how long requests have been waiting. Hence, while each of the N_r requesters is granted access $1/N_r$ of the times in the long run, one requester could suffer long starvation periods.

Instead, Random Permutations generates in every *arbitration (or permutation) window* a random permutation of all potential requesters. While the particular requester that is granted access in each arbitration round is random, each of the N_r requesters is granted access exactly once every arbitration window. For instance, for a resource shared across $N_r = 3$ requesters ($r1$, $r2$ and $r3$) each requester has 1/3 chances to be granted access first. If $r2$ is granted access, then $r1$ and $r3$ have 1/2 chances to be granted access second, whereas $r2$ cannot be granted access second. If $r3$ is granted access second, then $r1$ is automatically granted access third.

RP is implemented by creating a list where each requester appears once and sorting it randomly. Note that a requester could request access to the shared resource at any point in time w.r.t. the current arbitration window. Thus, the worst case occurs when, for instance, $r2$ arrives in the second slot of the current arbitration window, $r2$ was the first one in the window (so it just missed its opportunity), and has to wait for its slot in the next window, which, potentially, can be the last one. Recalling the example before, we could have the following arbitration windows: $< r2, r3, r1 >$, $< r3, r1, r2 >$, and $r2$ could arrive right after its slot in the first window has elapsed. Overall, in general the maximum number of slots a requester may have to wait is $2 \cdot (N_r - 1)$. This limits how long a request waits to be granted access.

RP can be applied with the same logic to the replacement policy. Instead of randomly choosing the way within the cache set that will be evicted next,

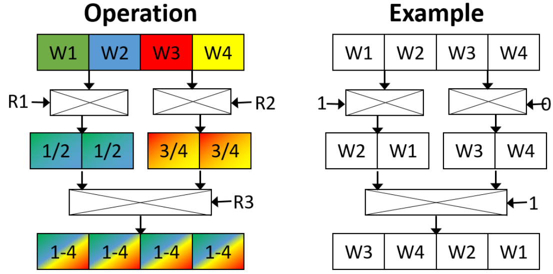


Figure 4: RP operation and an example for a 4-way cache.

RP generates a *random permutation window* per set in which the number of elements matches the number of cache ways W. Whether a line is evicted next is a random event (each line can be evicted with $1/W$ probability), but the eviction choices, though random, are not independent among them. In other words, each different permutation has the same probability to be created, each way number is in each of the W positions of the N_{Perm} different permutations N_{Perm}/W times and permutations are chosen (generated) randomly. However, given that each permutation contains each way number exactly once, it is impossible that a way is not selected for more than $2 \cdot (W-1)$ evictions, and a particular way can be evicted at most twice consecutively (if it is the last in one permutation and the first in the following one).

4.1.2. Implementing RP

For a W-way cache the total number of potential permutations of cache ways is $N_{perm} = W!$. At hardware level, implementing such an ideal solution could require $W! \cdot \lceil log_2(W) \rceil \cdot W$ bits for the permutations table, $\lceil log_2(W!) \rceil$ bits per set for the pointer that selects the permutation, and $\lceil log_2(W) \rceil$ bits for selecting the current permutation entry, plus the control logic for such an implementation. For a 4-way cache this would mean 192 bits for the table and 7 (5+2) bits per set. In an 8-way cache or higher this number significantly increases. In this section we implement a low-complexity solution that limits the number of potential permutations while preserving the properties of the ideal solution, and matching its average and WCET performance. We use area and logic as main metrics to assess the hardware feasibility of our approach.

Registers area: We implement RP by adding a bit vector per cache set, similar to that needed for Least-Recently Used (LRU) replacement. In the vector each way number is represented exactly once. Given a cache with W ways, this vector requires W fields with $\lceil log_2 W \rceil$ bits each, plus a pointer of $\lceil log_2 W \rceil$ bits to point to the current position in the vector. Thus, a 2-way cache requires 2+1 (vector+pointer) bits, a 4-way cache 8+2 bits and a 8-way cache 24+3 bits. For comparison purposes, LRU requires the same number of vector bits per set to keep the eviction order. Hence RP has similar area requirements in terms of bits to keep the replacement state as LRU, and only adds the bits of the pointer indicating the current position in the permutation.

Additional logic: The vector part of RP for a 4-way cache is depicted in Figure 4 (left). We denote the id assigned to cache ways as $w1$, $w2$, $w3$ and $w4$ respectively. Whenever the pointer wraps up, a new random permutation is generated. This is done, as shown in the picture, swapping different parts of the vector based on some random bits: $R1$, $R2$ and $R3$. $R1$ determines whether the first two elements are (randomly) swapped or not. $R2$ does the same for the last two elements. Finally, $R3$ determines whether the first pair of elements is swapped or not with the second pair. This simple implementation allows to generate a new permutation quickly and efficiently. LRU, instead, needs being able to extract any element from the list, place it at the beginning and shift all leftmost elements one position ($\lceil log_2 W \rceil$ bits) right. Thus, multiplexers (as for RP) and expensive parametric shifters are needed for LRU which compromises its scalability.

Random bits can be easily generated with a single low-cost linear feedback shift register [26, 29], which meets the requirements of MBPTA. Although each cache set has its own random permutation, the number of new permutations needed in one cycle in the cache is, at most, as many as cache ports exist since, in the worst case, all simultaneous accesses could produce a miss that requires a new random permutation in their respective cache sets. This would require 3 random bits per cache port simultaneously. The PRNG used has been proven capable to produce at least 32 bits per cycle if needed, thus making a PRNG able to feed multiple caches. On average, however, each cache port will require one new permutation every W cache misses, which occur seldom. Thus, usual random bit requirements per cycle are very low and a single PRNG could fit all caches in one or several cores.

One feature of our implementation is that it generates a subset of all potential permutations (8 of

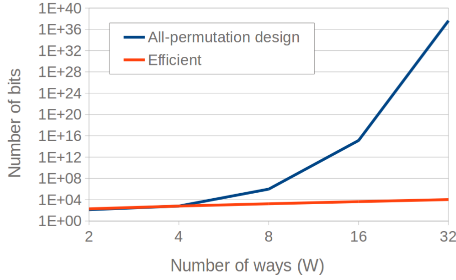


Figure 5: Hardware implementation cost for a 64-set cache of RP for different design choices.

the possible 24 in a 4-way cache). For instance, $w1$ and $w2$ can never be in separate pairs. This means that permutation $< w1, w3, w2, w4 >$ can never be produced. Yet our implementation still preserves the properties needed by MBPTA on random replacement: a given way occupies each position in the permutation with identical probabilities and where they are allocated is a purely random choice. The right side of Figure 4 shows an example of the generation of a new random permutation for the replacement of one cache set. Initially, we have the permutation $< w1, w2, w3, w4 >$. Given that random bit $R1 = 1$, $w1$ and $w2$ are swapped. Since $R2 = 0$, $w3$ and $w4$ are not swapped. Finally, $R3 = 1$, so $< w2, w1 >$ and $< w3, w4 >$ are swapped, leading to the new permutation $< w3, w4, w2, w1 >$.

In Figure 5 we compare the implementation costs of the fully randomized design and our cost efficient one of RP. For caches with 2 or 4 ways, the implementation cost is roughly the same. However, from 8 ways to 32 the cost of the ideal solution requires a million to 10^{37} bits respectively, while the efficient solution only needs 2,000 to 10,000 bits.

4.1.3. Controlling pathological scenarios

RP controls the pathological scenarios drawn for RR, i.e. ps1 and ps2, as presented next.

ps1: when W or fewer lines are (randomly) placed in the same set, they can replace each other a limited number of times. Let us recall the example in Section 3 where addresses A and B are accessed repeatedly. With RP two scenarios can occur (illustrated in Figure 6):

1. A and B are granted access with the same permutation window to take eviction decisions. In this case, A and B will be placed in different random ways.

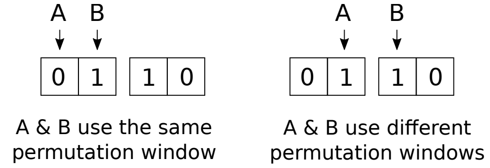


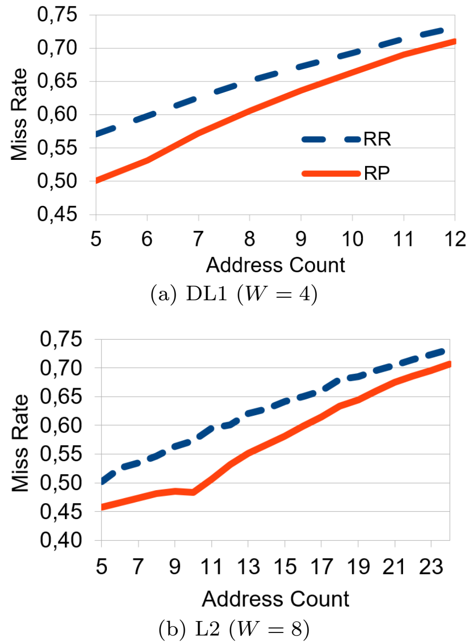
Figure 6: Example of the two different scenarios that can occur in ps1 with RP.

2. A evicts a line using the last element of one permutation window and B uses the first element of a new permutation window. In this case, again, two scenarios can occur. Firstly, A and B use different ways. And second, A and B are randomly mapped to the same way. In the latter case, B evicts A, but next time that A is fetched will necessarily use the same permutation window as B, so it will be placed in a different way. Thus, at most one mutual eviction will occur.

Overall, when 2 different addresses compete for the space in a given cache set, at most 1 mutual eviction will occur. In the general case, if K different addresses are accessed, where $K \leq W$, the worst case occurs when $K - 1$ addresses use the last elements of a permutation and the last address uses another permutation so that it evicts one of the other $K - 1$ addresses which, in turn, evicts another and so on and so forth. However, eventually the K addresses produce K consecutive evictions using elements of the same permutation, thus being placed in different cache ways and avoiding pathological evictions. Thus, the maximum number of pathological evictions can be expressed as:

$$N_{maxevict} \leq K - 1, \forall K \leq W \quad (2)$$

ps2: when the number of addresses mapped to a set exceeds the number of ways, i.e. $K > W$ compete for the same cache set, on every miss, RR can randomly evict recently touched lines hence decreasing temporal reuse. To show this we run an experiment in which we access a given number of addresses K in a sequence of size $2 \times K$ in which the addresses are randomly selected. We assume that all addresses are mapped to the same set and analyze the average hit rate of RR and RP when processing the same random sequence 1,000 times. Figure 7 shows the hit rate, for a 4-way cache (e.g. DL1-like) and a 8-way cache (e.g. L2-like), of RR and RP as K varies from $W + 1$ to $W * 3$. We



(a) DL1 ($W = 4$)

(b) L2 ($W = 8$)

Figure 7: Miss rate as a function of the number of accessed addresses for cache sizes similar to L1 and L2 respectively.

see that RP consistently lowers miss rates for both setups, with the benefit decreasing as the number of addresses increases w.r.t. the way size W since the overall miss rate naturally equalizes for high address counts, i.e. $K >> W$, when cache set capacity is largely exceeded.

4.2. Non-Most Recently Used Random Permutations (NMRURP)

NMRURP, similar to RP, limits pathological random replacement scenarios by increasing temporal reuse and enforcing random evictions to occur across all cache ways, but additionally, it guarantees that the MRU line cannot be evicted. In particular, it avoids potentially long mutual evictions by construction, thus behaving as RP in the example in Figure 3. Whenever the number of cache lines exceeds the space of the corresponding cache set, NMRURP has still some positive effects on reuse, but obviously the impact of replacement policies diminish as the set capacity is increasingly exceeded.

In order to introduce NMRURP, we build upon our other proposed randomized replacement policy, RP. First, we show how in specific cases RP may not favor locality sufficiently, and then how NMRURP improves over RP.

4.2.1. Locality awareness of RP

RP improves locality over RR by avoiding the replacement of a given cache line within a permutation (window). Hence, cache lines recently fetched cannot be evicted before crossing the boundary to the next window. However, there are two scenarios where RP may fail to preserve locality:

1. RP places no constraint across arbitration window boundaries. Hence, potentially, the same physical cache line could be evicted twice consecutively, which would allow the MRU line to be evicted. This occurs whenever a cache line is the last in a window and the first in the following window.
2. Also, RP does not keep any history on whether cache lines have been hit recently. Hence, if a given cache line can be the candidate for replacement according to RP, be hit, and then be evicted immediately due to a miss, thus causing an eviction of the MRU line.

Next, we illustrate those two scenarios with specific examples that serve the purpose to motivate the introduction of NMRURP.

RP example 1: window boundaries. Let us assume a 4-way cache whose current and next arbitration windows are $< w1, w2, w3, w4 >$ and $< w4, w2, w3, w1 >$ respectively. Let us further assume that the next eviction is dictated by the last slot of the current window ($w4$ in the first window), and $w4$ in the cache set contains cache line A. The sequence $B_1 A_1 B_2$, where the subscript only indicates access ordering to a given address, would cause 3 misses. First, B_1 would miss and would evict the line in $w4$, so address A. The pointer in the arbitration window would move to the first position in the next permutation, which is $w4$ again. Then A_1 would also miss, thus evicting B. Finally, B_2 would also miss and would replace the address in $w2$.

As shown, RP allows evicting the MRU cache line even if it has just been fetched when crossing arbitration window boundaries if the last slot of one window and the first slot of the following window randomly point to the same cache way, which occurs with probability $1/W$, where W is the number of cache ways.

RP example 2: MRU hit. Let us assume the same example, so the current arbitration window is $< w1, w2, w3, w4 >$, the candidate for eviction is $w4$, and A is stored in $w4$. In this case, the sequence $A_1 B_1 A_2$ would produce 2 misses since A_1 is a hit,

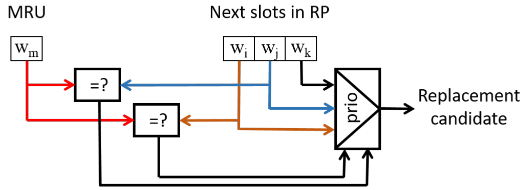


Figure 8: Additional logic for NMRURP w.r.t. RP.

thus becoming $w4$ (so A) the MRU line, B_1 is a miss and evicts A, and then A_2 also misses.

As shown, on a hit there are $1/W$ chances that the candidate for eviction is the cache line recently hit, so a subsequent cache miss may evict the MRU cache line with a non-negligible probability.

4.2.2. NMRURP replacement policy

NMRURP is a hybrid policy between NMRU and RP. In particular, it works as RP but, whenever the cache line to be evicted is the MRU, the following candidate in the arbitration window (or the first one in the next window if window boundaries are exceeded) is selected for eviction. This prevents, by construction, the eviction of the MRU cache line.

Let us recall **RP example 1** above. In this case, B_1 would evict A from $w4$, but A_2 would not be allowed to evict B since B is stored in the MRU way ($w4$). Hence, $w2$ would be replaced instead and access B_2 would hit. In the case of **RP example 2**, the behavior is similar. A_1 hits in $w4$, B_1 is not allowed to evict $w4$ and evicts $w2$, and A_2 is therefore a hit.

Overall, NMRURP increases locality w.r.t. RP in both cases, whenever the MRU was either hit or fetched due to a miss.

4.2.3. Implementing NMRURP

The implementation costs of NMRURP are slightly higher than those for RP. In particular, it requires a register of $\lceil log_2 W \rceil$ bits to store the identifier of the MRU cache line, two comparators to compare the current and next candidates for eviction with the MRU, and a priority decoder to select the appropriate candidate for eviction. Note that the output of the comparisons drive both, the priority decoder and the pointer shift in the arbitration windows to select the following eviction candidate. A schematic of the additional logic w.r.t. RP is depicted in Figure 8 for illustration purposes. As shown, only two slots need to be compared with the MRU since at most two consecutive slots may point to the same cache way given that each way has only one occurrence per window, and thus they can only repeat once across window boundaries. Thus, up to two slots may need to be bypassed, as it would be the case in **RP example 2** above.

Overall, the additional hardware cost is small and increasingly associative caches only require a slightly larger MRU pointer, thus justifying the scalability of this replacement policy.

5. MBPTA compliance

MBPTA requires that execution time distributions occurring during operation match or are upper-bounded by those enforced at analysis. This is achieved by means of time randomization or time upper-bounding [30]. In the particular case of replacement policies, this needs to be enforced too. In this section we review the MBPTA compliance of the different replacement policies discussed in this paper, namely LRU, NMRU, BT, RR, RP and NMRURP.

5.1. Cache controllability

In general, it is unaffordable predicting the cache state before running a program during operation, unless all accesses since the last flush are being tracked, which requires an unrealistic level of controllability. An explicit flush command could be issued before each software unit, however this would incur in a significant execution time and energy cost. Hence, cache flushing often occurs only across time partitions for the sake of memory consistency, but not across software units runs within a given time partition [7].

In this context, the most convenient solution to ensure worst-case cache effects are properly captured consists of enforcing an initial cache state at analysis time that leads to equal or higher execution times than any potential initial cache state that may occur during operation. In general, one would expect that the empty cache state provides this behavior, since no data is reused from previous runs and, consequently, more accesses should miss in cache. However, as shown in this section, this is not always the case for all replacement policies.

5.2. LRU

The LRU policy keeps the order in which the cache lines in a set have been last accessed. Therefore, two setups starting from different initial cache

states will reach the same state for a given cache set after W accesses to different cache lines in that cache set, where W stands for the number of cache ways.

This occurs because, whenever a cache line is accessed, whether it is a hit or a miss, it is promoted to the MRU position, and the remaining cache lines are shifted as needed to the LRU position to make room for this line be the MRU. If the access is a miss, all lines are shifted one position closer to the LRU position, and the LRU line is evicted. Conversely if the access is a hit, only those lines closer to the MRU position than the one hit are shifted towards the LRU position.

The only difference across two different initial cache (set) states with LRU relates to whether the first access to each of the first W different addresses accessed is a hit or a miss, which depends on the initial state.

Therefore, we can upper-bound the behavior during operation by doing one of the following things at analysis:

- Flush the cache before each run at analysis. This ensures that the first w accesses to different addresses in each set miss, and after those accesses the cache state is the same as that during operation regardless of the initial state during operation.
- Add the latency of w misses per set to the WCET estimate, which upper-bounds the gain obtained between the best and the worst initial cache states. This method could result in a more pessimistic outcome since we could be accounting for some gains that do not occur in practice because the program could also miss in the first w accesses to different addresses in each set.

If either of these two approaches is followed, the execution times of the program at analysis with the LRU replacement policy upper-bound those during operation and hence, LRU is MBPTA compliant. Note, however, that despite LRU its compliance with MBPTA requirements, it may still exhibit systematic pathological cases that, nevertheless, would already be captured, in the absence of timing anomalies, during the test campaign at analysis.

5.3. NMRU

NMRU policy selects the replacement based on a round-robin mechanism but protecting the MRU cache line in a set. Given two initial cache states, the MRU value in a specific set will be the same after just one access. The rest of the addresses, however, will depend on the initial state and the access sequence.

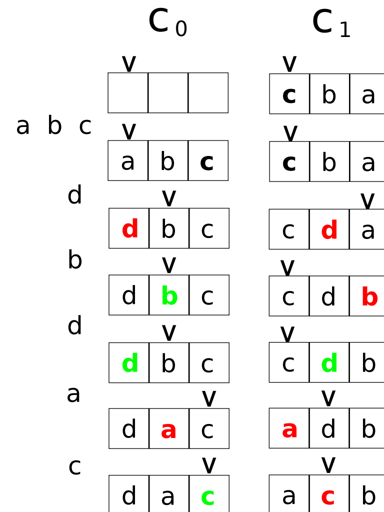


Figure 9: NMRU. Sequence that has more hits in the empty cache than in the one initialized.

The example in Figure 9 shows the same set for two different initial cache states c_0 and c_1. c_0 corresponds to the empty state, whereas c_1 has some contents. The pointer in each set indicates the next element to be replaced (following a round-robin policy). The bold cache line indicates the MRU line, so the one protected from eviction.

In this example, first, we make three accesses to fill the empty cache: a, b and c. Afterwards, we access a new cache line d that misses in both caches. Since c_0 and c_1 have different pointers and orders, they will evict different lines. c_0 will evict line a, whereas c_1 will evict line b, since the eviction pointer points to c that is protected (MRU line). The next access is to line c, which hits in c_0 and misses in c_1. After two more accesses that hit and then miss in both caches, the same case arises: an access to line c that hits in c_0 and misses in c_1.

As shown in this example, there is no guarantee that an empty initial cache state upper-bounds the execution time of a non-empty cache, and the execution time difference can be arbitrarily large since both initial states may not converge to the same state. Hence, we can claim that the empty cache state is not an acceptable initial state for analysis runs since it does not upper-bound all initial

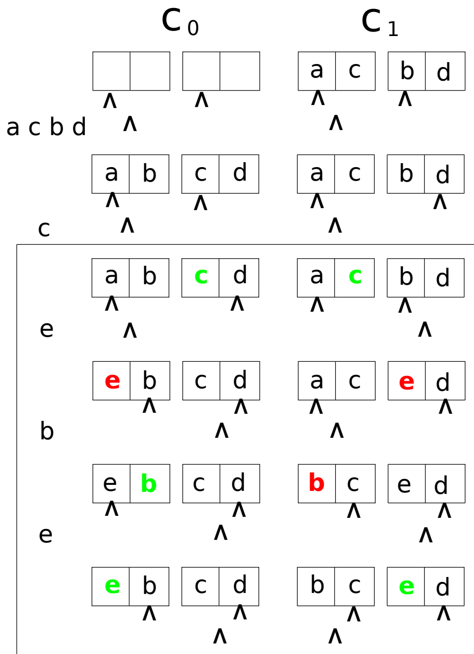


Figure 10: BT. Sequence that has more hits in the empty cache than in the one initialized.

cache states and access sequences. Moreover, our example already illustrates that specific access sequences may make any given initial state perform worse than another given state without converging to the same state, thus indicating that MBPTA compliance is not achieved for NMRU.

Only if we could enforce the same initial cache state at analysis and during operation, NMRU could be made MBPTA compliant. However, as explained before, the initial cache state during operation may not be controlled (e.g. flushed) in many cases.

5.4. BT

The Binary Tree (BT) replacement policy has an auxiliary tree structure that defines the state of the replacement algorithm for each cache set. The fact that cache lines are already stored in a particular location and such location, together with the access sequence, determines the replacement order, can lead to a pathological scenario where some accesses always miss. This can easily be seen with an example.

The example in Figure 10 shows a set of two different initial cache (set) states c_0 and c_1, one empty and one initialized respectively. The first 4 accesses, namely a, c, b and d are performed in both caches. In this example, the arrows point to the cache line to be replaced. Since the cache has 4 ways, we need 2 levels of arrows. The first level indicates the pair to be replaced first, and the second level what cache line inside the pair must be replaced. After these 4 accesses c_0 is filled (4 misses) and c_1 maintains its state (4 hits).

Another access to c occurs and hits in both caches. Then, we make a sequence of 3 accesses to cache lines e, b and e (all mapped to the same set). The first access misses on both caches, and the last hits on both. However, the second access (b) hits on the empty initial cache state, whereas it misses on the already initialized cache. The final state is equivalent to the third state shown in the example (after accessing c), but with the following conversions: $a' = e$, $b' = c$, $c' = b$, $d' = a$, $e' = c$, where the *prime* mark indicates the new state. This means that a sequence a', c', a' would again result in the same behavior: miss for both, then a hit for c_0 and a miss for c_1, and finally a hit for both. This pattern (shown in a box for illustration purposes), with the appropriate addresses, could repeat an arbitrarily long number of times, thus making the empty initial cache state lead to arbitrarily lower execution times than the non-empty state.

As for NMRU, given any initial cache state, we can devise an access sequence that makes a different initial cache state lead to systematically lower execution times. Hence, an initial state that upper-bounds all others does not exist. This implies that, measurements collected on an empty initial cache state do not upper-bound operation-time behavior, and differences across initial states cannot be upper-bounded in general. Because of this, we regard this cache policy as non MBPTA compliant.

5.5. RR

Random replacement (RR) policy chooses where to allocate a new cache line randomly. Hence, it does not keep any state on replacement order. With RR we cannot determine how many accesses to different addresses will make two different initial cache states reach the same state. However, since eviction choices are random and have uniform probabilities across lines, we can claim that whether accesses hit or miss does not alter cache replacement state (which is none for RR). Thus, any non-empty state leads, probabilistically, to equal or lower miss rates that an empty initial state.

This behavior can already be inferred from the work in [31]. In particular, authors prove that with

RR, given an initial cache state, causing random evictions can only lead to a probabilistically worse execution time. In our case, by causing an infinite number of evictions, we would reach the empty cache state that, therefore, would lead to a probabilistically worse execution time than any other initial state.

Hence, on a non-empty initial cache state during operation, we may experience additional hits and thus, lower execution times than those experienced at analysis with an empty cache state. We can conclude, therefore, that RR is MBPTA compliant with an empty initial cache state at analysis.

5.6. RP

RP generates a permutation that will replace all cache lines in a cache set in w replacements. However, a program can start its execution in the middle of a permutation, so there can be a scenario where we need to perform $(2 \cdot w) - 1$ replacements before all cache lines in the set have been replaced. This occurs when a given cache line is in the first slot of a window and in the last of the next window, and initially we start replacing the cache in the second slot. For instance, given the permutations $< w1, w2, w3, w4 >$ and $< w4, w2, w3, w1 >$, if the eviction pointer is in the second slot of the first permutation, we need 7 evictions to evict the line in $w1$, whereas if we are at the beginning of a permutation, we only need w replacements to evict all lines.

Let us build our argument on the MBPTA compliance of RP in two steps. First, we show that the difference between two empty initial cache states with different window alignment is up to $w - 1$ misses. Then, we show that for a non-empty cache state exists an empty cache state that upper-bounds the non-empty state. Such empty cache state is, by construction, up to $w - 1$ misses better than the worst empty cache state. Hence, at analysis we can enforce an empty initial cache state, where window alignment per set can be *any*, and increase WCET estimates $w - 1$ misses per set.

Difference across empty initial states. With RP there is a dependence between the position of the eviction slot of the current window and miss probabilities. In particular, given two initial empty cache states with different permutation window alignment, it may take up to $w - 1$ evictions until their eviction probabilities match (e.g. both align at the beginning of the permutation window), and from that point onwards, eviction probabilities are identical. Whether those up to $w - 1$ replacements lead an additional hit or miss each, depends on the access sequence. Hence, if we increase the WCET estimate by the impact of $w - 1$ misses per set, we can claim that RP is MBPTA compliant.

This can be better illustrated comparing RP with NMRU. Since MRU protection is a subcase of LRU, which is MBPTA compliant, let us consider only the round-robin part of NMRU for the sake of this discussion. NMRU fails to be MBPTA compliant because the eviction order of addresses is fixed (round-robin). Hence, the particular location of the pointer in the sets *determines* evictions. In the case of RP, evictions occur randomly and uniformly distributed across cache ways within a permutation window. Once two initial cache states reach the same window alignment, eviction probabilities are random and follow the same distribution across both states, thus having similar properties to those of RR. Hence, reaching such identical alignment (e.g. between the current window alignment and the worst potential alignment for the program under analysis) requires up to $w - 1$ evictions.

Difference between empty and non-empty states. Note that since cache hits do not alter RP state, a non-empty initial cache state c_1 can only lead to lower execution times that, at least, an empty initial cache state c_0. In particular, given an access a hitting in a preexisting line in c_1 and missing in c_0, the likelihood of a being evicted in c_0 is lower since the pointer moves to the following slot in the window. Eventually, this may make that a access to a is a hit in c_0 and a miss in c_1, thus producing the opposite effect. However, such extra miss in the non-empty cache state can only occur after an extra miss in the empty cache state, which guarantees that the empty cache state is probabilistically worse than the non-empty one.

Need for padding WCET estimates. So far we have shown that, theoretically, we may need to account for up to $w - 1$ extra misses per set. However, this holds under the assumption that the initial alignment is deterministic. However, we can break such dependence by randomizing the initial alignment with the permutation window both at analysis and during operation. We can enforce the flush process to choose randomly the window alignment in each set. At analysis, such flush is performed before each run. During operation, it is only needed at boot time. After that, the random alignment is modified by random choices for replacement, thus leading to a random alignment

before running the program under analysis, even if the number of evictions before its execution is deterministic (e.g. programs performing only cold misses). Therefore, RP provides MBPTA compliance by simply starting from an empty cache state with random window alignments in each set.

5.7. NMRURP

The case of NMRURP is analogous to that of RP, with the difference that the MRU line is protected. Hence, the difference in terms of permutation window alignments is up to $w-2$ lines. However, the MRU pointer may also differ across different initial states, so the maximum difference across empty initial cache states is again $w-1$ misses per set, as for RP. Also, the same reasoning that applies for empty and non-empty cache states for RP, applies for NMRURP, as well as the concept of enforcing a random window alignment on a cache flush.

Overall, by using an empty initial cache state and enforcing random window alignment in each set on a cache flush (flush must be used at boot time during operation), NMRURP can also be regarded as MBPTA compliant.

6. Evaluation

In this section, we evaluate RP and NMRURP in terms of average execution time and pWCET estimates, and compare them against LRU, NMRU, BT (average performance) and RR, LRU (average performance and pWCET).

6.1. Framework

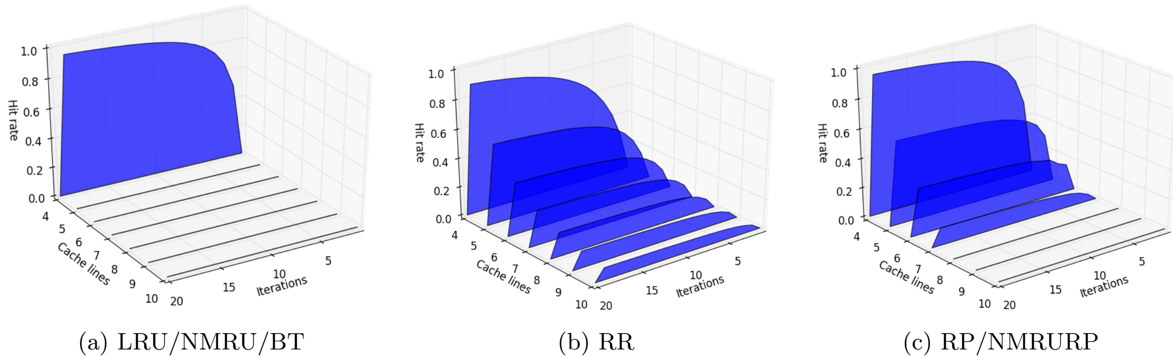
Processor model. We model a 4-core multicore processor with pipelined in-order cores with a cycle-accurate performance simulator based on the SoCLib simulation framework [32]. Our measurements are collected on one of the cores, while the remaining cores are idle. However, our setup is made MBPTA compliant as described in [30], thus meaning that arbitration on shared resources always occurs across all cores during our analysis, as if the remaining cores had pending requests. If another core is granted access to any shared resource in this analysis phase, the resource is kept busy for the maximum request duration. This way the contention considered during our analysis matches or upper-bounds that during operation regardless of what programs run in the other cores in practice.

The simulator has been configured to model the LEON4 multicore processor [33], against which it was assessed showing performance variations of around 1-3% only [34]. Each core includes L1 instruction (IL1) and data (DL1) caches, being DL1 write-through and write-allocate. Both caches are 16KB 4-way with 16B/line. A shared – yet partitioned – 512KB 4-way L2 cache is included so that each core has effectively a 128KB 1-way L2 cache, thus matching the setup of the LEON4 multicore processor. DL1 and IL1 hit and miss latencies are 1 cycle. On a miss, the shared bus is accessed with 4-cycles latency and then L2 latency is 2 cycles for both hits and misses. On a L2 miss, memory is accessed, whose latency is 16 cycles.

L1 caches implement random modulo placement [11], whereas L2 cache implements hash-based random placement [10], given that this has been shown a very convenient setup [11] for MBPTA. Note that, since we build upon a randomized placement policy, execution times change across runs, even if the replacement policy is deterministic as, for instance, in the case of LRU. By choosing a randomized placement policy, we enable MBPTA compliance for all those replacement policies also MBPTA compliant, and have a fair comparison across replacement policies since all of them build upon the same placement policy. In fact, we enforce the same set of random placements (e.g. the same set of 1,000 random placements for each of the 1,000 runs) across replacement policies so that the only source of variation across setups is the replacement policy. For average performance evaluation we consider that all caches use the same replacement policy, which can be either LRU, NMRU, BT, RR, RP or NMRURP. The L2 has no replacement policy since each core has a single L2 way, thus requiring no replacement policy. For pWCET estimation purposes, we compare our proposals, namely RP and NMRURP, with the existing MBPTA compliant replacement policies, namely RR and LRU. Note that, despite considering a multicore, evaluation is performed for programs in isolation since we focus on cache replacement effects.

Applications. We make a solid evaluation of our proposal with three different application setups.

- For illustration purposes, we consider a synthetic benchmark traversing a configurable number of times a vector with varying sizes ranging from 4KB to 40KB in 4KB steps, on a



(a) LRU/NMRU/BT (b) RR (c) RP/NMRURP

Figure 11: L1 data hit rate for the synthetic benchmark when varying the number of cache lines and iterations.

Table 1: Average results for the 10,000 executions and the worst 100 (1%) for the Mälardalen benchmarks.

	All						**Worst 1%**					
	LRU	**NMRU**	**BT**	**RR**	**RP**	**NMRURP**	**LRU**	**NMRU**	**BT**	**RR**	**RP**	**NMRURP**
Cycles	45,522	45,036	44,776	46,076	44,998	44,989	46,472	45,428	45,218	46,991	45,357	45,358
IL1 m.r.	0.017	0.017	0.017	0.018	0.017	0.017	0.018	0.018	0.018	0.019	0.018	0.018
DL1 m.r.	0.133	0.133	0.128	0.135	0.132	0.131	0.134	0.137	0.132	0.138	0.136	0.136
L2 m.r.	0.465	0.460	0.478	0.459	0.464	0.465	0.424	0.400	0.405	0.416	0.401	0.401

setup with modulo placement, so that we access in a round-robin fashion between 1 and 10 different lines in the same set. This allows us illustrating the different timing behavior for each replacement policy.

- We use a representative subset of the well-known Mälardalen Benchmark Suite [35]. We use 28 out of the 36 Mälardalen benchmarks, since we could not manage to run the remaining 8 in our simulation infrastructure.

- We also use a real railway application implementing a critical real-time function from the European Train Control System (ETCS) reference architecture. This application is in charge of controlling the safety functions related to the supervision of the train distance and speed. We use the 10 different input sets regarded as relevant by the end user (provider of the application), numbered from 0 to 9.

WCET estimation. We use MBPTA to derive pWCET estimates [6]. As cutoff probability for the pWCET estimates, we use 10^{-12} per run, so that a program executing up to 10,000 times per hour does not exceed a deadline miss rate of 10^{-8} per hour. Other cutoff probabilities show analogous trends.

All the statistical tests required for MBPTA applicability (i.e. independence and identical distribution tests [6]) were passed for all benchmarks, providing evidence that RP and NMRURP do not change the MBPTA compatibility of the underlying configuration.

6.2. Average performance

For these experiments, we use the same random seeds (and so, the same placements) for all configurations so that differences are produced due to the replacement policy. Miss rates as well as average execution time are obtained as the mean across all measurements for each input set and replacement policy.

6.2.1. Synthetic benchmark

Figure 11 analyzes the hit rate (y-axis) of the synthetic benchmark varying the number of cache lines accessed in round-robin (x-axis) and the number of iterations of the loop (z-axis).

- Figure 11(a) shows the matching results results for LRU, NMRU and BT replacement. We observe that a very high hit rate is obtained for up to 4 addresses accessed, which matches DL1 associativity. Above that point, all addresses are evicted systematically before being reused.

- In the case of RR (Figure 11(b)), we observe that it obtains decreasing hit rates as the number of addresses increases, but they are never

Table 2: Average results for the 10,000 executions and the worst 100 (1%) for the railway case study.

	All						**Worst 1%**					
	LRU	**NMRU**	**BT**	**RR**	**RP**	**NMRURP**	**LRU**	**NMRU**	**BT**	**RR**	**RP**	**NMRURP**
Cycles	3299	3288	3288	3311	3288	3289	3389	3306	3299	3408	3305	3306
IL1 m.r.	0.151	0.151	0.151	0.152	0.151	0.151	0.151	0.151	0.151	0.153	0.151	0.151
DL1 m.r.	0.302	0.302	0.302	0.305	0.302	0.302	0.303	0.314	0.307	0.312	0.315	0.314
L2 m.r.	0.781	0.781	0.781	0.776	0.781	0.781	0.794	0.772	0.776	0.783	0.772	0.772

zero. However, we also observe that hit rates slowly increase with the number of iterations for 4 addresses due to the cases where lines evict each other despite fitting in cache. A similar trend occurs for 2 and 3 addresses, but it is omitted in the plot since visually it is not so obvious.

- Finally, Figure 11(c) shows the matching results for RP and NMRURP. We observe that the hit rate grows rapidly with the number of iterations for 4 addresses. It also grows faster than RR for 2 and 3 addresses. However, for larger address counts the hit rate decreases faster than for RR being zero for 8 addresses. However, in that case the real problem is not the replacement policy, but the fact that cache capacity has been largely exceeded.

6.2.2. Mälardalen

Table 1 shows the average number of cycles and miss rates for IL1, DL1 and L2 for the 10,000 executions performed for each replacement policy. For the MBPTA compliant replacement policies, we also show the average of the 1% of simulations that had the highest execution times. The worst 1% runs for RP and NMRURP perform as good as the average of all runs for LRU and the other deterministic policies. As shown, for the highest 1% execution times, the gap between RR and RP/NMRURP increases due to the bounded pathological evictions with RP/NMRURP. Also, there is a significant gap with LRU, which also triggers some pathological cases in some sets for some placements, thus showing that RP and NMRURP are the best MBPTA compliant replacement policies. LRU only performs slightly better than RR but worse than RP/NMRURP. This occurs because it preserves temporal locality better than RR, but its pathological cases make it worse than RP/NMRURP. Note that L2 miss rates are lower for the worst 1% than on average for all runs. This relates to the fact that DL1 accesses dominate execution time, and higher execution times occur for higher DL1 miss rates. Thus, although the number of L2 misses remains barely constant for the worst 1%, the number of accesses increases and hence, the L2 miss rate decreases.

Differences between RP and NMRURP in terms of average performance are marginal for the evaluated benchmarks. While differences among them exist and may be relevant in some specific cases, most programs do not exhibit often those specific cases where NMRURP is superior to RP and hence, their average performance is roughly identical. Although NMRU and BT are not compatible with MBPTA, as discussed before, we also include them in this evaluation, showing that their performance, both across all measurements and across the worst 1%, is roughly identical to that of RP and NMRURP, which, however, attain MBPTA compliance.

For the sake of completeness, we have also considered 8-way caches, despite the target processor (LEON4 multicore) only implements 4-way caches. However, other embedded processors also implement 8-way L1 caches. Since the L1D cache has shown to be the most sensitive cache to the replacement policy used for this benchmark suite, we have only varied L1D cache set-associativity (using 8 instead of 4 ways). Results across all benchmarks barely changed, showing less than 1% variation w.r.t. the 4-way setup in terms of execution time. For instance, results for NMRURP with an 8-way DL1 are 44,789 cycles on average, and 45,071 cycles for the worst 1%. Since no further insight was observed, detailed results have been omitted.

6.2.3. Railway case study

For the railway case study, average results across input sets are shown in Table 2. As shown, RP and NMRURP provide small gains w.r.t. RR in terms of cycles, DL1 and IL1 miss rates. RP and NMRURP are slightly worse in terms of L2 miss rate. When compared against LRU and the other determinis-

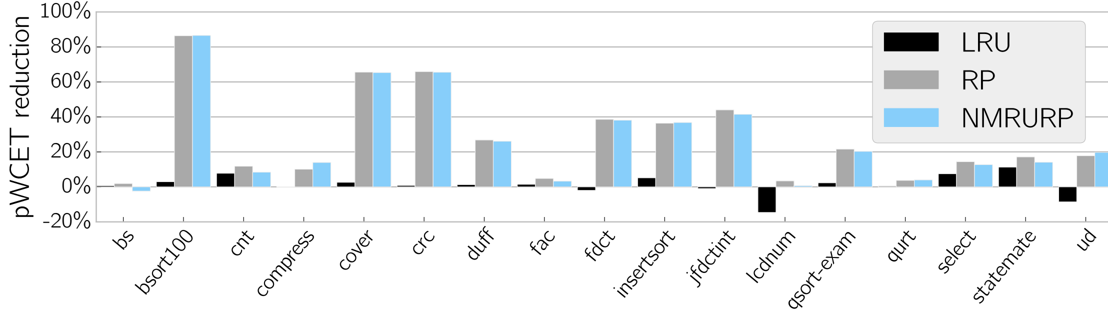


Figure 12: pWCET reduction ($p = 10^{-12}$) for the Mälardalen benchmarks w.r.t. RR.

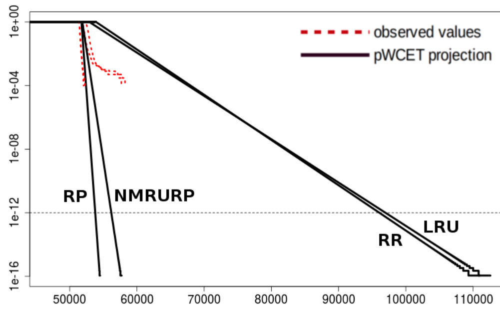


Figure 13: pWCET for `jfdc-tint` Mälardalen Benchmark for all MBPTA compliant replacement policies.

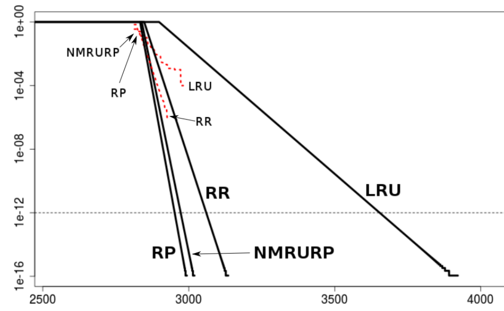


Figure 15: pWCET curve for input 9 of the railway case study w.r.t. RR.

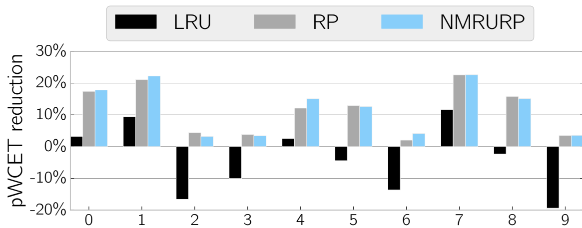


Figure 14: pWCET reduction ($p = 10^{-12}$) for the rail case study for all MBPTA compliant replacement policies.

tic policies, we also observe negligible differences. However, if we keep only the highest 100 measurements (the worst 1% of them), we observe that differences increase, evidencing that RR may produce pathological scenarios *ps1* and *ps2*, as we have further verified inspecting the sequences of events for the worst RR runs. Conversely, RP and NMRURP limit the maximum number of evictions so that its worst case is better than that of RR. As for Mälardalen, LRU performs slightly better than RR but worse than RP/NMRURP since LRU produces sporadic but significant pathological cases.

Analogously to the case of Mälardalen benchmarks, differences between RP and NMRURP in terms of average performance for the railway case study are marginal, and NMRU and BT, which are not MBPTA compliant, perform roughly as RP and NMRURP.

6.3. Worst-case performance

As shown in the previous section, the differences between RR/LRU and RP/NMRURP grow at the tail of the distribution (i.e. for the highest values): for instance, average execution time for RP is 1% lower than for RR and 3% lower for the 1% highest execution times of RR. The latter translates into tighter pWCET estimates for RP/NMRURP.

6.3.1. Mälardalen

For Mälardalen benchmarks, in Figure 12 we present the reduction of pWCET estimates at 10^{-12} w.r.t. those of RR. We observe that RP and NMRURP are consistently better than those for RR. The improvement ranges from 1% to 86%, being

24% on average for RP and NMRURP. This significant reduction in pWCET evidences the advantage of using RP or NMRURP instead of RR or LRU. LRU is on average 1% better than RR, performing a better on some cases and worse in others.

While discrepancies between RP and NMRURP are larger in terms of pWCET estimates than in terms of average performance, they are low across individual benchmarks and marginal on average. In fact, if we consider confidence intervals of 95% for pWCET estimates, intervals overlap for RP and NMRURP, thus indicating that differences are not statistically significant. Part of our future work consists of investigating whether specific patterns causing different behavior between RP and NMRURP exist to a sufficient extent in some industrial programs so that a better selection among RP and NMRURP replacement policies can be performed.

Figure 13 shows the pWCET distribution when using RR, LRU, RP and NMRURP for the `jfdctint` Mälardalen Benchmark. Red dotted lines and black straight lines represent the CCDF for the measured data and the pWCET curves respectively. RP and NMRURP provide increasingly higher gains as the exceedance threshold decreases due to the fact that RP and NMRURP avoid pathological evictions by construction. Since RR can produce some such pathological evictions with relevant probability, MBPTA accounts for that by smoothening the shape of the curve and shifting it to the right. Analogously, LRU can produce some pathological cases, which has also some significant impact on pWCET estimates.

6.3.2. Railway case study

For the rail application Figure 14 shows the pWCET estimates at 10^{-12} w.r.t. those of RR. We observe that the pWCET estimates of RP and NMRURP are consistently better than those for RR , while LRU is sometimes better and sometimes worse, although on average LRU performs worse since its pathological cases can occur systematically as opposed to those of RR, which occur with decreasing probabilities. RP pWCET reduction w.r.t. RR is 11% on average, reaching 22.6% for input set 7.

For illustration purposes, Figures 15 shows the pWCET distribution when using LRU, RR, NMRURP and RP for the railway case study (input set 9). Observed trends are similar to for Mälardalen benchmarks: the lower the exceedance probability considered, the larger the gap between RP/NMRURP and RR/LRU. Moreover, in this particular case, we observe that LRU is significantly worse than RR (almost 20% worse) due to the systematic nature of its pathological cases.

Overall, RP and NMRURP provide slightly better average performance than RR, and similar performance as deterministic policies. However, in terms of pWCET estimates, our proposed replacement policies, RP and NMRURP, are consistently better than RR and LRU by preserving locality and avoiding pathological cases by construction.

7. Related work

Literature on timing analysis is abundant and it is not the purpose of this work describing the different existing approaches. Instead, we refer the interested reader to a detailed survey describing pros and cons of static, measurement-based and hybrid timing analysis approaches [4]. Recently, existing timing analysis approaches together with those based on probabilistic analysis have also been compared taking into account their potential sources of unreliability [36].

Research on replacement policies is abundant, but often targets either improving average performance or achieving deterministic predictability. Among those we find FIFO and LRU replacement policies as well as enhanced versions of them such as protected LRU [37] and pseudo-LRU, which has already been deployed in some IBM processors [38]. A performance comparison of these replacement policies including also NMRU is presented in [39]. Also, an oracle replacement policy has been defined as a reference but not made implementable [40]. Work on optimizing replacement policies for second (L2) and third level (L3) caches is abundant [41, 42, 43, 44, 45]. Those works, either for uniform [41, 42, 43] or non-uniform [44, 45] cache access architectures, leverage the fact that L1 caches filter many accesses, so that access patterns in L2 and L3 caches differ noticeably from those in L1 caches. In general, those cache policies have systematic pathological cases due to their deterministic nature, thus being unfriendly for MBPTA, as it is the case for LRU.

Random replacement (RR) policies have already been used in some COTS processors such as the ARM Cortex R4 [46] and the PowerPC 7450 [47]. These policies, however, have not been shown to meet the degree of randomness needed by MBPTA since they usually build upon low-quality

PRNGs. Instead, a LEON-based multicore processor adapted to be MBPTA-compliant employs a high-quality PRNG, thus meeting MBPTA requirements [8]. However, those conventional random replacement policies, as shown in this paper, may lead to unbounded pathological eviction scenarios with decreasing probabilities, which impact pWCET estimates. Therefore, RP, by removing by construction those pathological scenarios, provides a significant advantage in terms of pWCET estimates in the context of MBPTA. In this work we have analysed replacement policies in the context of MBPTA. Other works [48, 49] analyse them for Static Timing Analysis.

8. Conclusions

MBPTA has been proven to be a powerful timing analysis approach enabling WCET estimation for complex software running on complex hardware. In particular MBPTA enables the use of arbitrarily complex cache hierarchies and placement/replacement policies. In this work we analyze the impact of cache replacement policies showing that deterministic ones can cause systematic pathological cases, thus degrading the quality of the WCET estimates. Conversely, random replacement makes pathological cases non systematic, but they can still occur with decreasing probabilities. This ultimately enforces MBPTA to account for some unfortunate cases with large number of random replacements.

We, then, propose two new randomized replacement policies, RP and NMRURP. We show that they completely remove pathological cases by preserving cache locality to some extent. This allows avoiding pathological cases and hence, improving WCET estimates drastically. Our evaluation on a set of benchmarks and a railway case study show that both policies largely outperform random replacement in terms of WCET estimates, despite average performance gains are rather modest. Whether one of the two randomized replacement policies proposed in this paper, namely RP and NMRURP, is superior to the other remains to be proven since differences among them in our evaluation cannot be regarded as statistically significant. Thus, as part of our future work we plan to verify whether large discrepancies among both policies can be found in other applications, as well as extend this analysis to more recently proposed replacement techniques [41, 43].

Acknowledgments

This work has been partially supported by the Spanish Ministry of Economy and Competitiveness (MINECO) under grant TIN2015-65316-P, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No. 772773) and the HiPEACH Network of Excellence. Pedro Benedicte and Jaume Abella have been partially supported by the MINECO under FPU15/01394 grant and Ramon y Cajal postdoctoral fellowship number RYC-2019-14717 respectively.

References

[1] P. Benedicte et al., RPR: A Random Replacement Policy with Limited Pathological Replacements, in: ACM/SIGAPP Symposium On Applied Computing, 2018.

[2] D. Buttle, ETAS GmbH, germany, real-time in the prime-time. keynote talk., in: ECRTS, 2012.

[3] J. Owens, Delphi automotive, the design of innovation that drives tomorrow. Keynote talk., in: DAC, 2015.

[4] R. Wilhelm et al., The worst-case execution-time problem overview of methods and survey of tools, ACM Transactions on Embedded Computing Systems 7 (2008) 1–53.

[5] L. Cucu-Grosjean et al., Measurement-based probabilistic timing analysis for multi-path programs, in: ECRTS, 2012.

[6] J. Abella et al., Measurement-based worst-case execution time estimation using the coefficient of variation, ACM Trans. Des. Autom. Electron. Syst. 22 (4) (2017) 72:1–72:29. doi:10.1145/3065924.

[7] F. Wartel et al., Timing analysis of an avionics case study on complex hardware/software platforms, in: DATE, 2015.

[8] P. Machado et al., Probabilistic timing analysis on time-randomized platforms for the space domain, in: DATE, 2017.

[9] E. Mezzetti and T. Vardanega, A rapid cache-aware procedure positioning optimization to favor incremental development, in: RTAS, 2013.

[10] L. Kosmidis et al., A cache design for probabilistically analysable real-time systems, in: DATE, 2013.

[11] C. Hernandez et al., Random modulo: a new processor cache design for real-time critical systems, in: DAC, 2016.

[12] L. Kosmidis, R. Vargas, D. Morales, E. Quiñones, J. Abella, F. J. Cazorla, TASA: Toolchain Agnostic Software Randomisation for Critical Real-Time Systems, in: ICCAD, 2016.

[13] L. Kosmidis et al., Probabilistic timing analysis on conventional cache designs, in: DATE, 2013.

[14] COBHAM, LEON3 Processor. Probabilistic platform, http://www.gaisler.com/index.php/products/processors/leon3.

[15] Esterel Technologies, SA, Efficient Developement of Safe Avionics Software with DO-178B Objectives Using SCADE Suite - Methodological Handbook (2006).

[16] International Organization for Standardization, ISO/DIS 26262. Road Vehicles – Functional Safety (2009).

[17] International Electrotechnical Comission, IEC 61508, Functional Safety of Electrical/Electronic/Programmable Electronic Safety-related Systems, Edition 2.0 (2009).

[18] G. Bernat, A. Colin, S. M. Petters, Wcet analysis of probabilistic hard real[hyphen]time systems, in: In Proceedings of the 23rd Real[HYPHEN]Time Systems Symposium RTSS 2002, 2002, p. 279[HYPHEN][HYPHEN]288.

[19] G. Bernat, A. Colin, S. Petters, pwcet: a tool for probabilistic worst[hyphen]case execution time analysis of real[hyphen]time systems, Tech. rep. (2003).

[20] Z. Stephenson, J. Abella, T. Vardanega, Supporting industrial use of probabilistic timing analysis with explicit argumentation, in: INDIN, 2013.

[21] S. Kotz, S. Nadarajah, Extreme value distributions: theory and applications, World Scientific, 2000.

[22] K. Palma Silva, L. Arcaro, R. Silva de Oliveira, On using gev or gumbel models when applying evt for probabilistic wcet estimation, 2017. doi:10.1109/RTSS.2017.00028.

[23] G. Lima, D. Dias, E. Barros, Extreme value theory for estimating task execution time bounds: A careful look, 2016, p. 200[HYPHEN]211. doi:10.1109/ECRTS.2016.20.

[24] J. Abella et al., Heart of Gold: Making the improbable happen to extend coverage in probabilistic timing analysis, in: ECRTS, 2014.

[25] K. Kedzierski, M. Moretó, F. J. Cazorla, M. Valero, Adapting cache partitioning algorithms to pseudo-lru replacement policies, in: 24th IEEE International Symposium on Parallel and Distributed Processing, IPDPS 2010, Atlanta, Georgia, USA, 19-23 April 2010 - Conference Proceedings, 2010, pp. 1–12. doi:10.1109/IPDPS.2010.5470352.
URL https://doi.org/10.1109/IPDPS.2010.5470352

[26] I. Agirre et al., IEC-61508 SIL 3 compliant pseudo-random number generators for probabilistic timing analysis, in: DSD, 2015.

[27] J. Jalle et al., Bus designs for time-probabilistic multicore processors, in: DATE, 2014.

[28] K. Lahiri, A. Raghunathan, G. Lakshminarayana, LOTTERYBUS: a new high-performance communication architecture for system-on-chip designs, in: Proceedings of the 38th annual Design Automation Conference, DAC '01, 2001, pp. 15–20.

[29] P. Alfke, Efficient Shift Registers, LFSR Counters, and Long Pseudo-Random Sequence Generators, Xilinx (1996).

[30] L. Kosmidis et al., Fitting processor architectures for measurement-based probabilistic timing analysis, Elsevier Journal of Microprocessors and Microsystems 47 (B) (2016) 287–302.

[31] L. Kosmidis, E. Quiones, J. Abella, T. Vardanega, F. J. Cazorla, Achieving timing composability with measurement-based probabilistic timing analysis, in: 16th IEEE International Symposium on Object/component/service-oriented Real-time distributed Computing (ISORC 2013), 2013, pp. 1–8. doi:10.1109/ISORC.2013.6913193.

[32] SoCLib, -, http://www.soclib.fr/trac/dev (2003-2012).

[33] Cobham Gaisler, Quad Core LEON4 SPARC V8 Processor - LEON4-NGMP-DRAFT - Data Sheet and Users Manual (2011).

[34] J. Jalle et al., Validating a timing simulator for the NGMP multicore processor, in: DASIA, 2016.

[35] J. Gustafsson et al., The Mälardalen WCET benchmarks-past, present and future, in: WCET Workshop, 2010.

[36] J. Abella et al., WCET analysis methods: Pitfalls and challenges on their trustworthiness, in: SIES, 2015.

[37] R. Karedla, J. S. Love, B. G. Wherry, Caching strategies to improve disk system performance, Computer 27 (3) (1994) 38–46.

[38] T. Chen, P. Liu, K. Stelzer, Implementation of a pseudo-LRU algorithm in a partitioned cache, uS Patent number 7,069,390 (2006).

[39] H. Al-Zoubi, A. Milenkovic, M. Milenkovic, Performance evaluation of cache replacement policies for the spec cpu2000 benchmark suite, in: Proceedings of the 42Nd Annual Southeast Regional Conference, ACM-SE 42, ACM, New York, NY, USA, 2004, pp. 267–272. doi:10.1145/986537.986601.
URL http://doi.acm.org/10.1145/986537.986601

[40] L. A. Belady, A study of replacement algorithms for a virtual-storage computer, IBM Systems Journal 5 (2) (1966) 78–101.

[41] M. Qureshi et al., Adaptive insertion policies for high performance caching, in: ISCA, 2007.

[42] M. Chaudhuri, Pseudo-LIFO: The foundation of a new family of replacement policies for last-level caches, in: MICRO, 2009.

[43] A. Jaleel et al., High performance cache replacement using re-reference interval prediction (RRIP), in: ISCA, 2010.

[44] S. Bartolini, P. Foglia, C. A. Prete, Exploring the relationship between architectures and management policies in the design of NUCA-based chip multicore systems, in: Future Generation Computer Systems, 2018.

[45] A. Scolari, D. B. Bartolini, M. C. Santambrogio, A Software Cache Partitioning System for Hash-Based Caches, in: ACM Transactions on Architecture and Code Optimization (TACO), 2018.

[46] ARM, Cortex-R4 and Cortex-R4F Technical Reference Manual (2006).

[47] Freescale Semiconductor, MPC7450 RISC Microprocessor Family Reference Manual. Rev. 5, Freescale Semiconductor (2005).

[48] J. Reineke, Randomized caches considered harmful in hard real[hyphen]time systems, Leibniz Transactions on Embedded Systems 1 (1) (2014) 03:1[HYPHEN]03:13. doi:10.4230/LITES[HYPHEN]v001[HYPHEN]i001[HYPHEN]a003.
URL http://ojs.dagstuhl.de/index.php/lites/article/view/LITES[HYPHEN]v001[HYPHEN]i001[HYPHEN]a003

[49] S. Hahn, D. Grund, Relational cache analysis for static timing analysis, in: Real[HYPHEN]Time Systems (ECRTS), 2012 24th Euromicro Conference on, IEEE, 2012, p. 102[HYPHEN][HYPHEN]111.