Randomized Caches Considered Harmful in Hard Real-Time Systems

Jan Reineke

Saarland University
Saarbrücken, Germany
`reineke@cs.uni-saarland.de`

Abstract

We investigate the suitability of caches with randomized placement and replacement in the context of hard real-time systems. Such caches have been claimed to drastically reduce the amount of information required by static worst-case execution time (WCET) analysis, and to be an enabler for measurement-based probabilistic timing analysis. We refute these claims and conclude that with prevailing static and measurement-based analysis techniques caches with deterministic placement and least-recently-used replacement are preferable over randomized ones.

2012 ACM Subject Classification Computer systems organization∼Real-time system architecture, Theory of computation∼Caching and paging algorithms, Hardware∼Safety critical systems
Keywords and phrases Real-time systems, caches, randomization, WCET analysis
Digital Object Identifier 10.4230/LITES-v001-i001-a003
Received 2013-12-04 **Accepted** 2014-05-28 **Published** 2014-06-10

1 Introduction

Recent work has promoted the use of randomized caches in hard real-time systems [4, 20, 22, 23, 21, 7, 5, 25]. Along with randomized microarchitectures, this line of work proposes static probabilistic timing analysis (SPTA) and measurement-based probabilistic timing analysis (MBPTA). Caches are a major challenge in the timing analysis of traditional, deterministic microarchitectures. A key feature of randomized microarchitectures are caches with randomized placement and replacement. Such caches have been claimed to drastically reduce the amount of information required by WCET analyses. To quote Kosmidis et al. [23]: "The key benefit of embracing PTA (probabilistic timing analysis) is that execution timing becomes dramatically less dependent on execution history, with drastic reduction in the amount of information required to obtain tight WCET estimates in comparison to other timing analysis approaches."

In this paper, we critically assess these claims both in the context of static and measurement-based analysis. Specifically, we compare the precision of static cache analyses for caches with *least-recently-used* (LRU) replacement and with randomized replacement provided the *same* amount of information, i.e. the information stated to be sufficient for the analysis of randomized caches. Among deterministic caches we restrict our attention to those with LRU replacement, as it is widely considered to be the most predictable replacement policy, and it has been demonstrated to be efficiently implementable [1, 8]. Our analysis demonstrates that, with simple, state-of-the-art analyses, deterministic LRU replacement is preferable over random replacement. We also observe that, with its current restrictions, MBPTA is equally applicable to LRU caches as it is to randomized ones.

Regarding random placement, we show that it is impossible to assign non-zero hit probabilities to individual memory accesses that are *independent* of the outcome of other accesses. This means that caches with random placement are *not* amenable to the prevailing SPTA approach that relies on independence, as execution time profiles (ETPs) of individual instructions are convolved.

Finally, we provide a class of memory access sequences that is problematic for MBPTA under random placement. On these sequences, which may occur in practice due to loops, MBPTA either fails, is incorrect, or highly imprecise.

We provide the necessary background about probabilistic timing analysis in Section 2. Then, we introduce deterministic and randomized caches in Section 3. We assess the suitability of random placement and replacement for use in hard real-time systems in Sections 4 and 5. Finally, we briefly summarize our findings in Section 6.

2 Static and Measurement-Based Timing Analysis

The goal of static and measurement-based timing analysis for deterministic microarchitectures is to compute tight upper bounds on the worst-case execution times (WCET) of programs. The goal of timing analyses for randomized microarchitectures is slightly different: in such microarchitectures, very high execution times are possible, but—hopefully—only with a very low probability. Thus, timing analyses for such microarchitectures compute *exceedance functions*. These exceedance functions determine upper bounds on the probability of exceeding any given execution time. From such a function, and a probability threshold p, an execution time can be obtained that is exceeded only with a probability of e.g. $p = 10^{-12}$.

2.1 Static Probabilistic Timing Analysis

The de facto standard approach to static timing analysis (STA) for deterministic microarchitectures divides analysis into two main parts [31]:

1. *Low-level analysis*, which determines execution-time bounds for basic blocks (or other small contiguous program fragments) based on an accurate model of the underlying microarchitecture.
2. *Path-level analysis*, which determines an upper bound on the execution time of the program as a whole based on constraints on the control flow, e.g. loop bounds, and the execution-time bounds for basic blocks determined by low-level analysis.

A critical assumption of this approach is that the bounds obtained for a basic block during low-level analysis hold for *all possible* execution histories leading to the respective basic block. As execution times may depend heavily on the execution history, low-level analysis is often made context sensitive, e.g. by distinguishing the first iteration of a loop from the following ones.

While so far less studied and thus less developed, static probabilistic timing analysis (SPTA) follows a similar approach [4]:

1. For each instruction in the program, an *execution time profile* (ETP), i.e., a discrete probability distribution over the instruction's possible execution times, is derived. This step corresponds to the low-level analysis in STA.
2. To arrive at an ETP for a sequence of instructions the ETPs of all instructions in the sequence are combined by convolution. If multiple different execution paths are possible, their ETPs can be merged conservatively [4]. This roughly corresponds to path-level analysis in STA. From an ETP, a corresponding exceedance function can then be determined easily.

A critical assumption for SPTA to be sound is that the ETPs derived in step one are *independent* of each other. Only if they are independent, can they be soundly combined by convolution to arrive at an ETP for a sequence of instructions.

2.2 Measurement-based Probabilistic Timing Analysis

Measurement-based probabilistic timing analysis (MBPTA) derives exceedance functions for the execution time of a program from measurements. MBPTA as described by Cucu et al. [5] is based on Extreme Value Theory (EVT). In this approach, a series of end-to-end execution-time measurements is performed. The measurement results are used to estimate the parameters of an extreme value distribution, the Gumbel distribution. Measurements and estimation of the Gumbel distribution are interleaved until the distribution is considered to have converged [5]. The thus obtained Gumbel distribution then immediately induces an exceedance function.

Applicability of this approach relies on two assumptions:

1. The execution-time measurements can be modeled by independent and identically-distributed (i.i.d.) random variables.
2. The maximum of a sample of these i.i.d. random variables converges in distribution to the Gumbel distribution.

To satisfy the first assumption, Cucu et al. [5] propose a number of changes to the microarchitecture to eliminate the dependence of execution times on input data. For instance, input-dependent memory accesses must bypass the cache. They also initially limit their approach to single-path programs, which they later [5] show how to relax.

The satisfaction of the second assumption is validated during the analysis of a particular program by statistical tests.

3 Deterministic and Randomized Caches

Caches are fast but small memories that store a subset of the main memory's contents to bridge the latency gap between the CPU and main memory. To reduce management overhead and to profit from spatial locality, data is not cached at the granularity of words, but at the granularity of so-called *memory blocks*. To this end, main memory is logically partitioned into the set of equally-sized memory blocks $\mathcal{B} = \{0, \ldots, n\}$. Blocks are cached as a whole in *cache lines* of the same size. The size of a memory block varies from one processor to another, but is usually between 32 and 128 bytes.

When accessing a memory block, the cache logic has to determine whether the block is stored in the cache (a *cache hit*) or not (a *cache miss*). To enable an efficient look-up, each memory block can only be stored in a small number of cache lines. For this purpose, caches are partitioned into equally-sized *cache sets*. The size of a cache set is called the *associativity* k of the cache.

The *placement policy* determines the cache set a memory blocks maps to. In Section 3.2 we describe common deterministic and randomized placement policies.

Since caches are usually much smaller than main memory, a *replacement policy* must decide which memory block to replace upon a cache miss. In Section 3.1 we describe common deterministic and randomized replacement policies.

The performance of a cache depends on the *temporal* and *spatial locality* of the memory accesses. In Section 3.3, we describe two notions of locality that are approximated by state-of-the-art static (probabilistic) cache analyses.

3.1 Replacement Policies

Usually, replacement policies treat each cache set separately, so that accesses to a particular cache set do not influence replacement decisions in other cache sets. While exceptions to this rule exist, they have been identified as particularly unsuitable for real-time systems [17]. Thus, in the following, we only consider replacement policies treating each cache set separately.

Well-known *deterministic* replacement policies in this class are *least-recently used* (LRU), used in various Freescale processors such as the MPC603E and the TriCore17xx, as well as the recent Kalray MPPA 256; *pseudo-LRU* (PLRU), a cost-efficient variant of LRU, used in the

Freescale MPC750 family and multiple Intel microarchitectures; *most-recently used* (MRU), also known as *not most-recently used* (NMRU), another cost-efficient variant of LRU, used in the Intel Nehalem; *first-in first-out* (FIFO), also known as Round Robin, used in several ARM and Freescale processors such as the ARM922 and the Freescale MPC55xx family

Logically, LRU orders cached memory blocks by the recency of their last use, from most- to least-recently-used. Upon a miss, the least-recently-used block is evicted. Among deterministic policies, LRU is generally accepted as the most predictable policy [28]. Thus, in the following, among deterministic policies, we restrict our attention to LRU, which has been shown to be efficiently implementable [1], and which is used in the Kalray MPPA 256 [8] for predictability.

Quiñones et al. [25, 20, 21, 4] have promoted the use of *randomized* caches in real-time systems. They have focused on a policy, which we will call Random in the following, that was introduced by Belady [2]. Upon a miss, Random chooses the block to evict randomly and uniformly among the k cache lines of the cache set. Thus, upon a miss, a cached block—in the cache set that the accessed block maps to—is evicted with probability $\frac{1}{k}$. This policy is also referred to as *evict-on-miss* in the literature [7].

Several commercial processors are claimed to employ random replacement, e.g. the ARM720T, the ARM940T, the ARM11xx, and the Freescale MPC7450. However, most processor documentations are inconclusive about the exact meaning of "random". Such caches could be based on hardware random number generators that generate random numbers from a physical process, such as thermal noise, or they could employ deterministic pseudo-random number generators. The well-documented MPC7450 [18] allows to choose between two random policies for its second-level caches [10]: "The simpler one uses a modulo counter that is incremented on each clock cycle and whose value determines the cache line to replace."

To achieve independence between cache-miss probabilities, Cazorla et al. [4] have also proposed the *evict-on-access* policy, which evicts a block uniformly at random upon each memory access, rather than upon each cache miss. In the following, we limit our attention to Random, as it provably dominates *evict-on-access* in terms of the induced exceedance function on any workload.

Randomized policies have been studied extensively in the context of competitive analysis [3]. Policies such as Mark and Equitable have been shown to have smaller competitive ratios than *any* deterministic policy. These results concern the *expected performance* of a policy, rather than the performance achieved with high probability, which would be of greater interest in the hard real-time setting. However, recent results by Komm et al. [19] suggest that randomized policies can also be shown to be competitive "with high probability".

3.2 Placement Policies

A placement policy can be formalized as a mapping from memory blocks to cache sets: $place : \mathcal{B} \to \{0, \ldots, s-1\}$, where s is the number of cache sets.

The most common *deterministic* placement policy for set-associative caches is *modulo* placement: $place_{modulo}(b) = b \bmod s$. The number of sets s is usually a power of two, so that the cache set of a block is simply determined by its $\log s$ least significant bits.

In *random* placement the mapping $place_{random}$ from memory blocks to cache sets is chosen randomly from the set of all mappings $\mathcal{B} \to \{0, \ldots, s-1\}$. For static-analysis purposes it is convenient if the mapping is chosen uniformly at random from this set. Then, the probability of block b mapping to cache set t, $P(place_{random}(b) = t)$, is $\frac{1}{s}$. Note, that $place_{random}$ needs to be *fixed* for the entire execution of a program. Otherwise, it would not be possible to locate memory blocks that were cached earlier under a different mapping.[1]

[1] Changing the mapping at runtime requires either flushing of cache contents or their migration.

Kosmidis et al. present an approximation of random placement in hardware [20, 21] based on a parametric hash function and in software [22] on top of conventional caches with deterministic placement. The crucial difference between the hardware and the software solution is that the software solution can only randomize mapping at the granularity of “memory objects”, i.e., memory entities normally stored in consecutive memory addresses such as functions, basic blocks, or arrays.

3.3 Notions of Locality

Caches rely on *locality* in the memory access sequences generated by programs. Different ways of capturing locality have been proposed over time. Two notions of locality particularly relevant to LRU and Random replacement are the *reuse distance* and the *stack distance* of a memory access.

The *reuse distance* of a memory access to block b is the number of memory accesses between the current and the previous access to block b. The first access to a block has reuse distance ∞. As an example, we have annotated each memory access in the following sequence with its reuse distance:

$$a^{\infty}, b^{\infty}, b^{0}, a^{2}, c^{\infty}, d^{\infty}, d^{0}, c^{2}, b^{5}, a^{5}.$$

In contrast to the reuse distance, the *stack distance* of an access to block b is defined as the number of *distinct* memory blocks accessed between the current and the previous access to block b. The stack distance of a block is sometimes also referred to as the *age* of the block. The first access to a block has stack distance ∞. In the sequence from above, the stack distances are as follows:

$$a^{\infty}, b^{\infty}, b^{0}, a^{1}, c^{\infty}, d^{\infty}, d^{0}, c^{1}, b^{3}, a^{3}.$$

Note that the stack distance of any access is less than or equal to its reuse distance.

We will see later how hit and miss probabilities of a memory access can be given based on its reuse and stack distance for both randomized and deterministic caches.

3.4 Cache Analysis in Static (Probabilistic) Timing Analysis

Cache analysis is an important part of low-level analysis. In STA, its purpose is to classify memory accesses in the program as either definite hits or definite misses. Sometimes, an access may result in a hit *or* a miss depending on the execution history leading to the access. As a consequence of such inherent uncertainty or uncertainty due to analysis imprecision, cache analysis may also classify an access as “unknown”. Due to timing anomalies [24, 29] it is not always safe to simply assume a cache miss in case of uncertainty.

Similarly, in SPTA [4], a probability needs to be attached to the hit and the miss case for each memory access. Current SPTAs assume microarchitectures in which hits and misses have a fixed, context-independent cost and thus cache-related timing anomalies may not occur. As a consequence, it is sufficient to determine a lower bound h on the hit probability of an access, which induces an upper bound of $1 - h$ on its miss probability. Together with hit and miss latencies $hit_{latency}$ and $miss_{latency}$ we get the following ETP for a memory instruction with hit probability h:

$$\begin{pmatrix} hit_{latency} & miss_{latency} \\ h & 1-h \end{pmatrix}.$$

The issue of timing anomalies in the pipeline is orthogonal to that of using deterministic or randomized caches. In order not to mix the two issues, we compare deterministic[2] and randomized caches in the context of SPTA, i.e., in terms of deriving lower bounds on the hit probability of a memory access.

[2] Caches with LRU replacement do not exhibit timing anomalies.

4 Deterministic versus Random Replacement in Fully-Associative Caches

4.1 In Static Probabilistic Timing Analysis

In a cache with LRU replacement, an access b with stack distance $sd(b)$ less than the associativity k is a hit, otherwise it is a miss:

$$P(hit_{LRU}(b)) = \begin{cases} 1 & : sd(b) < k \\ 0 & : sd(b) \geq k \end{cases} \tag{1}$$

Note, that the hit probabilities of different memory accesses in a sequence are *independent*. Static cache analyses thus determine upper bounds on the stack distance of each memory access to guarantee cache hits. Such analyses are called *must analyses* [9]. Analogously, *may analyses* [9] determine lower bounds on stack distances to guarantee cache misses.

In the case of a fully-associative cache there are two challenges for may and must analyses:

1. The initial state of the cache is unknown. Thus, must analyses conservatively assume an upper bound $> k$ on the stack distance of any block at program start. Similarly, may analyses assume a lower bound of 0.
2. Logically, memory accesses are at the granularity of words, not memory blocks. Thus a *value analysis* needs to determine for pairs of memory accesses whether they refer to the same memory block or not.[3] This is trivial for instruction caches, but may be very hard for input-dependent data accesses.

In a cache with Random replacement the situation is different. In contrast to LRU, the hit probability of an access cannot be given purely in terms of its stack or reuse distance. Zhou [32] observes that the hit probability of an access b to a block that has been accessed before is

$$P(hit_{Random}(b)) = \left(1 - \frac{1}{k}\right)^{m} \tag{2}$$

where m is the number of cache misses between access b and the previous access to the same memory block. Clearly m is bounded from above by b's reuse distance $rd(b)$, so

$$P(hit_{Random}(b)) \geq \left(1 - \frac{1}{k}\right)^{rd(b)} \tag{3}$$

This formula correctly underestimates the hit probability of an individual access. Unfortunately, however, hit probabilities computed with the formula above are *not* independent of each other. Thus, the convolution of corresponding ETPs may underestimate the probability of observing a given number of misses. Consider, e.g., the access sequence a, b, c, a, b, c and a cache with associativity 2. Clearly, *at least one* miss must occur on the final three accesses of the sequence, as the first access to c will evict either a or b. Yet, the convolution of the ETPs obtained from Equation (3) yields a non-zero probability of having *no* misses on those three accesses, because the hit probability of each individual access is greater than zero[4]. As a consequence, the probability of observing four or more cache misses on the entire sequence, which is 1, would be underestimated.

[3] Note, that it is not necessary to determine which memory block is referred to by a memory access. It is sufficient to determine for pairs of accesses whether they refer to the same block or not. This is exploited by *relational cache analysis* [16].

[4] According to Equation (3), it is $\frac{1}{4}$.

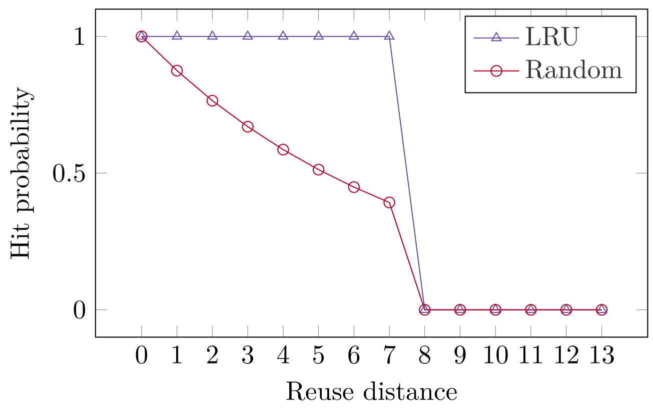

Figure 1 Lower bounds on hit probabilities of memory accesses for LRU and Random in terms of reuse distances for a cache of associativity 8 based on Equations 1 and 4.

Davis et al. [7] provide the following formula, which bounds the hit probability of an access b, *independently* of whether preceding accesses hit or miss, in terms of the reuse distance of b:

$$P(hit_{Random}(b)) \geq \begin{cases} \left(1-\frac{1}{k}\right)^{rd(b)} & : rd(b) < k \\ 0 & : rd(b) \geq k \end{cases} \tag{4}$$

The more optimistic formula for hit probabilities given in [20] has been refuted in [6]. Based on the reuse distance, the above formula is the most precise hit probability that holds independently of the outcome of previous memory accesses.

For associativity 8, Figure 1 illustrates the hit probabilities of LRU and Random in terms of reuse distances. Remember that by definition the stack distance $sd(b)$ of an access is less than or equal to its reuse distance $rd(b)$. With this in mind, comparing the hit probabilities for LRU and for Random from Equations 1 and 4, we make the following two observations:

▶ Observation 1. With the *same information* about an access sequence, i.e. upper bounds on the reuse distances of accesses, the hit probabilities for LRU are always greater than or equal to the hit probabilities for Random.

▶ Observation 2. In case of LRU, and in contrast to Random, current cache analyses can profit from bounding stack distances, which can be arbitrarily lower than reuse distances.

▶ Conclusion 1. With simple, state-of-the-art analysis methods, LRU replacement is preferable over Random replacement in static (probabilistic) timing analysis.

In general, we note that the state (or the state distribution in case of randomized caches) of *any* cache is a function of the history of memory accesses. This holds independently of whether the cache is deterministic or randomized. More precisely, the state of any fully-associative cache depends, at least, on the suffix of the history of memory accesses containing k distinct blocks, where k is the associativity of the cache. Among all policies, randomized or deterministic, the state of an LRU-controlled cache depends on the *shortest* suffix of the access history. Thus, LRU requires the least information about the access history to fully determine its state [28].

Other Deterministic Policies: FIFO, PLRU, and MRU. Common deterministic policies such as FIFO, PLRU, and MRU have been found to be less predictable than LRU [28]. In contrast to LRU, the exact hit probability of an access cannot be determined in terms of stack or reuse distances for these policies. The best lower bounds on hit probabilities that can be given based on stack distances for FIFO, PLRU, and MRU are:[5]

$$P(hit_{FIFO}(b)) \geq \begin{cases} 1 & : sd(b) < 1 \\ 0 & : sd(b) \geq 1 \end{cases} \tag{5}$$

$$P(hit_{PLRU}(b)) \geq \begin{cases} 1 & : sd(b) < \log_2 k + 1 \\ 0 & : sd(b) \geq \log_2 k + 1 \end{cases} \tag{6}$$

$$P(hit_{MRU}(b)) \geq \begin{cases} 1 & : sd(b) < 2 \\ 0 & : sd(b) \geq 2 \end{cases} \tag{7}$$

Static (probabilistic) timing analyses that are based purely on stack (or reuse) distances thus yield worse results under FIFO than under Random, MRU, PLRU, or LRU. MRU and PLRU are incomparable with Random. Depending on the benchmark, and the resulting distribution of reuse distances, MRU and PLRU may yield better results than Random and vice versa.

It should be noted that there are access sequences (and programs that generate such sequences) on which Random outperforms LRU and other deterministic policies with a very high probability. A prime example for such sequences are so-called “payroll sequences”, i.e. sequences of the form $(a_1, \ldots, a_{k+1})^*$, where k is the associativity of the cache. On such sequences LRU incurs cache misses only. Similar sequences can be constructed for *every* deterministic policy.

To profit from Random replacement in such cases, more sophisticated analysis techniques are required. Such analyses will likely have to derive conditional hit probabilities and combine results for individual memory accesses in a different way than convolution, which is not required in case of LRU. Similarly, sophisticated static analyses have recently been proposed for FIFO [11, 12, 15], PLRU [13], and MRU [14]. Yet, while being more complex than Ferdinand’s analysis for LRU [9], they still do not quite achieve the same level of precision.

Stack Distance versus Reuse Distance. As an example where reuse and stack distances may differ a lot, consider instruction accesses following the execution of a small, nested loop:

```
x = 0
y = 0
for i in [1, 1000]:
    for j in [1, i]:
        x = x+1
    y = y+1
```

Assume for simplicity that the instructions implementing each line of the program occupy exactly one memory block. Thus, the program’s instructions occupy exactly six memory blocks. Then, except for the first access, the stack distance of each instruction access for `y = y+1` is three. Yet, the corresponding reuse distance depends on the value of `i`. In the last iteration of the outer loop the reuse distance of the instruction access for `y = y+1` is 2001.

In practice, the relation between reuse and stack distances likely varies strongly within benchmarks and from one benchmark to another. Unfortunately, we were not able to find empirical data concerning their relation, with one exception: Sen and Wood [30] plot the distribution of reuse distances for accesses of a given stack distance (Figure 3 in [30]) for an online transaction processing application.

[5] This follows immediately from the competitiveness of the respective policies relative to LRU [26].

4.2 In Measurement-Based Probabilistic Timing Analysis

MBPTA derives WCET estimates from a series of execution-time measurements. Cache performance depends on the initial state of the cache and on the sequence of memory accesses provided to the cache. Cucu et al. [5] flush the cache upon program start and eliminate all input-dependent memory accesses from the program. Thus, on a given path through the program, the sequence of memory accesses is the same for different program inputs. Then, a number of end-to-end execution-time measurements is performed on each program path. By nature of the approach, the analysis results apply only to those program paths for which measurements have been performed.

In such a scenario, i.e., flushed cache and no input-dependent memory accesses, the requirements of MBPTA, namely independence and identical distribution of execution times on a given program path, are also met by a conventional cache with LRU replacement: the cache behavior will be identical on each measurement; it follows a degenerate probability distribution. Independence is thus trivial. In fact, the same argument applies to *any* deterministic cache replacement policy.

Thus, under the conditions described above, a *single* measurement will reveal the worst case—in terms of cache performance. This compares with having to perform hundreds of measurements in case of a randomized cache [5].

▶ Conclusion 2. Deterministic replacement yields more efficient MBPTA than Random replacement.

For LRU, the empty state is the worst-case initial state for any memory access sequence [27]. Therefore, measurements obtained starting with a flushed cache yield upper bounds on the number of cache misses under any initial cache state for LRU. In this case, flushing the cache would only be required during the measurement-based analysis and could in principle be disabled during normal operation, assuming the microarchitecture features no timing anomalies [24, 29].

5 Deterministic versus Random Placement in Set-Associative Caches

5.1 In Static Probabilistic Timing Analysis

In a set-associative cache with s cache sets, the placement policy partitions the stream of memory accesses into s substreams, each of which is processed by one of the s cache sets. For set-associative caches, it is convenient to define the reuse and the stack distance of an access based on the subsequence the access belongs to. In other words, the *reuse distance* of an access to block b is the number of memory accesses between the current and the previous access to block b within the *same* cache set. Let the *stack distance* be defined analogously for set-associative caches.

Then we get the same hit probabilities in terms of stack and reuse distance for LRU and Random as in case of a fully-associative cache.

The additional difficulty in static cache analysis with *deterministic placement* is thus to determine which memory accesses map to the same cache set. This is again trivial for instruction accesses, but may be very difficult for data accesses. Note, however, that it is *not* required to determine the absolute cache set an access maps to, as demonstrated by relational cache analysis [16].

Randomized placement [20, 21] promises to reduce the analysis effort for set-associative caches, as two memory blocks will only collide in the cache with a certain probability. If the placement function is chosen randomly from a uniform distribution over all possible placement functions, then the probability of any two blocks to map to the same cache set is $\frac{1}{s}$. Based on this assumption, Kosmidis et al. [20] derive the following hit probability for a direct-mapped cache in terms of the stack distance[6] of an access:

$$P(hit_{Random}(b)) = \left(1 - \frac{1}{s}\right)^{sd(b)} \tag{8}$$

This formula is correct. However, as in the case of Zhou's formula for random replacement, hit probabilities determined in this way are *not* independent. Consider the following access sequence:

$a, b, a, b, a, b, a, b, a, b$

As the placement is chosen randomly at program start and does not change during program execution, there are only two possibilities: either a and b systematically collide in the cache or they do not. They collide with probability $\frac{1}{s}$. Thus with a probability of $\frac{1}{s}$ all ten memory accesses will be cache misses, and with a probability of $1 - \frac{1}{s}$ only two (compulsory) misses will occur. In the example, each access's miss probability is $1 - (1 - \frac{1}{s})^1 = \frac{1}{s}$. Assuming independence of these miss probabilities would incorrectly yield a probability of $\frac{1}{s^8} \ll \frac{1}{s}$ of observing ten misses.

Unfortunately, *no* non-zero hit probability can be assigned to an access based on its stack distance that is *independent* of other whether previous accesses resulted in hits or misses. Yet, independence is required by the current SPTA approach. To see this, consider arbitrarily long sequences of the form $a, b, a, b, a, b, \ldots$ No matter how long the sequence, with a probability of $\frac{1}{s}$ all accesses will miss in a direct-mapped cache. For any non-zero hit probability p assigned to each individual access there is a length n of the sequence, such that the convolution of the ETPs based on p will underestimate the probability of incurring n misses.

▶ Observation 3. In case of random placement, no mutually independent hit probabilities greater than zero can be assigned to individual memory accesses with stack distances greater than zero.

Observation 3 immediately implies the following conclusion:

▶ Conclusion 3. Random placement requires complex static analyses that take into account conditional hit probabilities. Random placement is thus not amenable to current analysis approaches.

5.2 In Measurement-Based Probabilistic Timing Analysis

As we have seen in the previous section, with random placement there are cases in which we observe very few misses with a high probability p and very many misses with a low probability $1 - p$, with *no* cases in between the two extremes.

If p is sufficiently close to 1, MBPTA is unlikely to ever observe the case of very many misses. Then, its observations are indistinguishable from a case in which many misses are in fact impossible. Consider as an example[7] the same sequence $\sigma_{slow} = a, b, a, b, a, b, \ldots$ as above, which may be generated by a loop, and a very large direct-mapped cache with $s = 10^6$ cache sets. The probabilities of the two possible execution times are depicted in Figure 2.

Even $10000 = 10^4$ measurements will only reveal the worst case with a probability of $1 - (1 - 1/10^6)^{10^4} < 1\%$. In other words, with a probability greater than 99%, all measurements will yield exactly two cache misses, and thus the sequence would be indistinguishable from the sequence $\sigma_{fast} = a, b, b, b, b, \ldots$, which will yield exactly two misses independently of the placement.

[6] Here, $sd(b)$ refers to b's stack distance among *all* memory accesses, i.e., *not* to its stack distance among blocks mapping to the same cache set.

[7] While this example is slightly construed, due to the unrealistically high number of cache sets, the sequence $a, b, c, d, a, b, c, d, \ldots$ in a 3-way set-associative cache with a more realistic $s = 10^2$ leads to similar results, yet is more difficult to analyze precisely.

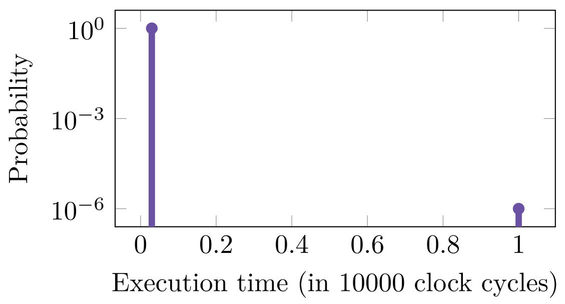

Figure 2 Execution-time distribution on example sequence with random placement.

MBPTA is used to estimate execution times that are only exceeded with a very low probability, such as 10^{-12}, ideally without performing 10^{12} measurements. In our example, with a probability of $10^{-6} \gg 10^{-12}$, the execution time for sequence σ_{slow} will be very high, as all memory accesses will result in cache misses.

MBPTA bases its estimates solely on measurement results. It would thus generate the *same* execution-time distribution for programs that generate the sequences σ_{slow} and σ_{fast} with a high probability. In such a situation there are three possibilities:

1. MBPTA correctly estimates the execution-time distribution for the sequence σ_{slow}.
2. MBPTA incorrectly underestimates the execution-time distribution for the sequence σ_{slow}.
3. Based on the measurement results, the statistical tests in MBPTA reject such programs.

The two latter cases are clearly undesirable. In the first case, MBPTA's estimate would have to be the same for the sequence σ_{fast}. However, a correct estimate for σ_{slow} is necessarily extremely pessimistic for σ_{fast}, which never exhibits more than two cache misses. This leads us to our final conclusion:

▶ Conclusion 4. Random placement is not suitable for MBPTA.

6 Summary

We have critically assessed the suitability of randomized caches for use in hard real-time systems. We observe that when used in SPTA, state-of-the-art cache analyses deliver better hit probabilities for LRU than for Random replacement with the *same* amount of information. With the restrictions currently imposed upon the use of randomized caches in MBPTA, i.e., no input-dependent memory accesses, deterministic caches with LRU replacement can also be safely employed in MBPTA. This comes with the additional benefit of requiring only a single measurement to identify the worst-case cache performance.

Further, we have shown that non-trivial hit probabilities under random placement are *not* independent and can thus not be safely combined by convolution. We have also identified simple access sequences, which may be generated by simple loops, on which MBPTA must—by construction—be either unsound, extremely pessimistic, or fail to produce an estimate at all.

Despite the negative results obtained in this paper, we believe that randomization might have a place in microarchitectures for real-time systems. A benefit over deterministic microarchitectures may be an increase in *robustness*. It is future work to rigorously analyze the benefits of randomization in this direction.

Acknowledgements. I would like to thank the anonymous reviewers for their helpful remarks. This work was supported by the Deutsche Forschungsgemeinschaft as part of the Transregional Collaborative Research Centre SFB/TR 14 (AVACS).

References

1 Bryan D. Ackland, Alex Anesko, Douglas M. Brinthaupt, Steven J. Daubert, Asawaree Kalavade, Joseph Knobloch, E. Micca, Mallik Moturi, Chris J. Nicol, Jay H. O'Neill, Joseph H. Othmer, Eduard Säckinger, Kanwar J. Singh, J. Sweet, Christopher J. Terman, and Joseph Williams. A single-chip, 1.6 billion, 16-b mac/s multiprocessor DSP. *IEEE Journal of Solid-State Circuits*, 35(3):412–423, March 2000. doi:10.1109/4.826824.

2 Laszlo A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966. doi:10.1147/sj.52.0078.

3 Allan Borodin and Ran El-Yaniv. *Online computation and competitive analysis*. Cambridge University Press, New York, NY, USA, 1998.

4 Francisco J. Cazorla, Eduardo Quiñones, Tullio Vardanega, Liliana Cucu, Benoit Triquet, Guillem Bernat, Emery Berger, Jaume Abella, Franck Wartel, Michael Houston, Luca Santinelli, Leonidas Kosmidis, Code Lo, and Dorin Maxim. PROARTIS: Probabilistically analyzable real-time systems. *ACM Transactions on Embedded Computing Systems*, 12(2s):94:1–94:26, May 2013. doi:10.1145/2465787.2465796.

5 Liliana Cucu-Grosjean, Luca Santinelli, Michael Houston, Code Lo, Tullio Vardanega, Leonidas Kosmidis, Jaume Abella, Enrico Mezzetti, Eduardo Quiñones, and Francisco J. Cazorla. Measurement-based probabilistic timing analysis for multi-path programs. In *24th Euromicro Conference on Real-Time Systems, ECRTS'12, Pisa, Italy*, pages 91–101, Washington, DC, USA, July 2012. IEEE Computer Society. doi:10.1109/ECRTS.2012.31.

6 Robert I. Davis. Improvements to static probabilistic timing analysis for systems with random cache replacement policies. In *2013 4th Real-Time Scheduling Open Problems Seminar, RTSOPS'13*, July 2013.

7 Robert I. Davis, Luca Santinelli, Sebastian Altmeyer, Claire Maiza, and Liliana Cucu-Grosjean. Analysis of probabilistic cache related pre-emption delays. In *25th Euromicro Conference on Real-Time Systems, ECRTS'13, Paris, France*, pages 168–179, July 2013. doi:10.1109/ECRTS.2013.27.

8 Benoît Dupont de Dinechin, Duco van Amstel, Marc Poulhiès, and Guillaume Lager. Time-critical computing on a single-chip massively parallel processor. In *Design, Automation & Test in Europe Conference & Exhibition, DATE'14, Dresden, Germany*, pages 513–518, March 2014. doi:10.7873/DATE2014.110.

9 Christian Ferdinand and Reinhard Wilhelm. Efficient and precise cache behavior prediction for real-time systems. *Real-Time Systems*, 17(2-3):131–181, 1999. doi:10.1023/A:1008186323068.

10 Daniel Grund. *Static Cache Analysis for Real-Time Systems – LRU, FIFO, PLRU*. PhD thesis, Saarland University, 2012. URL: https://www.epubli.de/shop/buch/Static-Cache-Analysis-for-Real-Time-Systems-Daniel-Grund-9783844216998/13092.

11 Daniel Grund and Jan Reineke. Abstract interpretation of FIFO replacement. In *Static Analysis, 16th International Symposium, SAS'09, Los Angeles, CA, USA*, pages 120–136. Springer, August 2009. doi:10.1007/978-3-642-03237-0_10.

12 Daniel Grund and Jan Reineke. Precise and efficient FIFO-replacement analysis based on static phase detection. In *22nd Euromicro Conference on Real-Time Systems, ECRTS'10, Brussels, Belgium*, pages 155–164. IEEE Computer Society, July 2010. doi:10.1109/ECRTS.2010.8.

13 Daniel Grund and Jan Reineke. Toward precise PLRU cache analysis. In *10th International Workshop on Worst-Case Execution Time Analysis, WCET'10, Brussels, Belgium*, pages 23–35. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Germany, July 2010. doi:10.4230/OASIcs.WCET.2010.23.

14 Nan Guan, Mingsong Lv, Wang Yi, and Ge Yu. WCET analysis with MRU caches: Challenging LRU for predictability. In *2012 IEEE 18th Real Time and Embedded Technology and Applications Symposium, Beijing, China*, pages 55–64. IEEE, April 2012. doi:10.1109/RTAS.2012.31.

15 Nan Guan, Xinping Yang, Mingsong Lv, and Wang Yi. FIFO cache analysis for WCET estimation: a quantitative approach. In *Design, Automation and Test in Europe, DATE'13, Grenoble, France*, pages 296–301. EDA Consortium San Jose, CA, USA / ACM DL, March 2013. URL: http://dl.acm.org/citation.cfm?id=2485362.

16 Sebastian Hahn and Daniel Grund. Relational cache analysis for static timing analysis. In *24th Euromicro Conference on Real-Time Systems, ECRTS'12, Pisa, Italy*, pages 102–111. IEEE Computer Society, July 2012. doi:10.1109/ECRTS.2012.14.

17 Reinhold Heckmann, Marc Langenbach, Stephan Thesing, and Reinhard Wilhelm. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE*, 91(7):1038–1054, 2003. doi:10.1109/JPROC.2003.814618.

18 Freescale Semiconductor Inc. MPC7450 RISC microprocessor family reference manual, rev. 5, 2005.

19 Dennis Komm, Rastislav Kràlovic, Richard Kràlovic, and Tobias Mömke. Randomized online algorithms with high probability guarantees. In *31st International Symposium on Theoretical Aspects of Computer Science, STACS'14, Lyon, France*, pages 470–481. Schloss Dagstuhl – Leibniz-Zentrum für Informatik, March 2014. doi:10.4230/LIPIcs.STACS.2014.470.

20 Leonidas Kosmidis, Jaume Abella, Eduardo Quiñones, and Francisco J. Cazorla. A cache design for probabilistically analysable real-time systems. In *Design, Automation and Test in Europe, DATE'13, Grenoble, France*, pages 513–518. EDA Consortium San Jose, CA, USA / ACM DL, March 2013. URL: http://dl.acm.org/citation.cfm?id=2485416.

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

cache designs for probabilistically analysable real-time systems. *IEEE Transactions on Computers*, 99(PrePrints):1, 2013. doi:10.1109/TC.2013.182.

22 Leonidas Kosmidis, Charlie Curtsinger, Eduardo Quiñones, Jaume Abella, Emery D. Berger, and Francisco J. Cazorla. Probabilistic timing analysis on conventional cache designs. In *Design, Automation and Test in Europe, DATE'13, Grenoble, France*, pages 603–606. EDA Consortium San Jose, CA, USA / ACM DL, March 2013. URL: http://dl.acm.org/citation.cfm?id=2485435.

23 Leonidas Kosmidis, Eduardo Quiñones, Jaume Abella, Tullio Vardanega, and Francisco J. Cazorla. Achieving timing composability with measurement-based probabilistic timing analysis. In *2013 16th IEEE Symposium on Object/Component/Service-oriented Real-time Distributed Computing, ISORC'13*, 2013.

24 Thomas Lundqvist and Per Stenström. Timing anomalies in dynamically scheduled microprocessors. In *20th IEEE Real-Time Systems Symposium, Phoenix, AZ, USA*, pages 12–21. IEEE Computer Society, December 1999. doi:10.1109/REAL.1999.818824.

25 Eduardo Quiñones, Emery D. Berger, Guillem Bernat, and Francisco J. Cazorla. Using randomized caches in probabilistic real-time systems. In *21st Euromicro Conference on Real-Time Systems, ECRTS 2009, Dublin, Ireland*, pages 129–138. IEEE Computer Society, July 2009. doi:10.1109/ECRTS.2009.30.

26 Jan Reineke. *Caches in WCET Analysis.* PhD thesis, Universität des Saarlandes, 2008. URL: http://rw4.cs.uni-saarland.de/~reineke/publications/DissertationCachesInWCETAnalysis.pdf.

27 Jan Reineke and Daniel Grund. Sensitivity of cache replacement policies. *ACM Transactions on Embedded Computing Systems*, 12(1s):42, 2013. doi:10.1145/2435227.2435238.

28 Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, 2007. doi:10.1007/s11241-007-9032-3.

29 Jan Reineke, Björn Wachter, Stephan Thesing, Reinhard Wilhelm, Ilia Polian, Jochen Eisinger, and Bernd Becker. A definition and classification of timing anomalies. In *6th International Workshop on Worst-Case Execution Time Analysis, WCET'06, Dresden, Germany.* Schloss Dagstuhl – Leibniz-Zentrum für Informatik, March 2006. doi:10.4230/OASIcs.WCET.2006.671.

30 Rathijit Sen and David A. Wood. Reuse-based online models for caches. In *ACM SIGMETRICS / International Conference on Measurement and Modeling of Computer Systems, SIGMETRICS'13, Pittsburgh, PA, USA*, pages 279–292. ACM, June 2013. doi:10.1145/2465529.2465756.

31 Reinhard Wilhelm, Jakob Engblom, Andreas Ermedahl, Niklas Holsti, Stephan Thesing, David B. Whalley, Guillem Bernat, Christian Ferdinand, Reinhold Heckmann, Tulika Mitra, Frank Mueller, Isabelle Puaut, Peter P. Puschner, Jan Staschulat, and Per Stenström. The worst-case execution-time problem – overview of methods and survey of tools. *ACM Transactions on Embedded Computing Systems*, 7(3), 2008. doi:10.1145/1347375.1347389.

32 Shuchang Zhou. An efficient simulation algorithm for cache of random replacement policy. In *Network and Parallel Computing, IFIP International Conference, NPC'10, Zhengzhou, China*, pages 144–154. Springer, 2010. doi:10.1007/978-3-642-15672-4_13.