

# Using Randomized Caches in Probabilistic Real-Time Systems

Eduardo Quiñones  
Barcelona Supercomputing Center  
Barcelona, Spain  
e-mail: eduardo.quinones@bsc.es

Guillem Bernat  
Rapita Systems  
York, England  
e-mail: bernat@rapitasystems.com

Emery D. Berger  
Dept. of Computer Science  
University of Massachusetts, Amherst, USA  
e-mail: emery@cs.umass.edu

Francisco J. Cazorla  
Barcelona Supercomputing Center  
Barcelona, Spain  
e-mail: francisco.carzorla@bsc.es

**Abstract**—While hardware caches are generally effective at improving application performance, they greatly complicate performance prediction. Slight changes in memory layout or data access patterns can lead to large and systematic increases in cache misses, degrading performance. In the worst case, these misses can effectively render the cache useless. These pathological cases, or “cache risk patterns”, are difficult to predict, test or debug, and their presence limits the usefulness of caches in safety critical real-time systems, especially in hard real-time environments.

In this paper, we explore the effect of randomized cache replacement policies in real-time systems with stringent timing constraints. We present simulation-based results on representative examples that illustrate the problem of performance anomalies with standard cache replacement policies. We show that, by eliminating dependencies on access history, randomized replacement greatly reduces the risk of these cache-based performance anomalies, enabling probabilistic worst-case execution time analysis.

**Index Terms:** *hard real-time systems; randomized hardware; cache replacement policies; timing analysis*

## I. INTRODUCTION

In safety-critical hard real-time systems, such as flight control systems, engine management systems or satellite control systems, guaranteeing that all computations meet their deadlines—computing their *worst-case execution time* (WCET)—is essential [1]. Numerous approaches exist to perform WCET analysis; see Wilhelm et al. for a survey of methods and tools [2].

The need for more complex and demanding safety-critical real-time systems implies that future processors for such systems are likely to resemble current high-performance processors. These processors have numer-

ous features designed to improve performance, such as pipelines, multiple levels of caches, and multiple cores. Unfortunately, the increased complexity of these modern hardware architectures makes computing the WCET even more difficult. On older CPUs, computing the WCET was relatively simple because each machine instruction ran for a fixed number of cycles. However, systems with more advanced processor features can lead to enormous variations in instruction execution and memory access times, greatly complicating WCET computation.

One key processor feature that exemplifies this problem is the cache. On one hand, caches are ubiquitous in most microprocessors because they can dramatically improve application performance. By storing recently-accessed data items in high-speed memory close to the processor, caches exploit locality in memory access patterns and can reduce access times by up to two orders of magnitude [3].

On the other hand, while caches often improve performance, they do not do so reliably or predictably. The performance of caches depends both on recent access history and the memory addresses of accessed objects, making it difficult to predict the access time to memory for any given object at any point in time. Memory access times are even harder to predict in the face of complicated cache hierarchies with multiple levels of caches. Previous work studying the difficulties of using caches in critical real-time systems shows that small program changes that lead to different memory layouts can trigger *pathological* cache behavior: systematic cache misses that lead to large increases in worst-case execution times [4].

The difficulty of predicting memory access times in the presence of caches—even though the program may never trigger pathological behaviors—can lead to worst-case execution time estimates that are extremely pessimistic, where each unpredictable access is assumed to be a cache miss. This complexity has led some real-time systems to disable caches entirely.

### Contributions

The thesis of this paper is that, instead of moving towards building hardware and software that are more *predictable* and therefore have systematic pathological worst-case scenarios, we should move towards a more truly randomized behavior that is guaranteed by design to exhibit pathological worst-case behavior only with an extremely low probability. In this paper, we assess the impact of one aspect of randomization, a randomized replacement policy for the instruction cache.

We first show that the potential worst-case impact of pathological cache behavior is large. We present experimental evaluations on microbenchmarks that show that these pathological memory layouts are likely for cache line replacement policies like LRU and set placement policies using both standard *set-associative* and *skewed-associative* caches [5], and that they lead to dramatic increases in cache misses. We then show that randomized cache replacement dramatically reduces the probability of systematic cache misses, probabilistically reducing the worst-case number of cache misses without substantially degrading performance. Finally, we argue that randomization enables probabilistic timing analysis because of the *statistical independence* of random variables, allowing quantification of the likelihood of a cache risk pattern.

The remainder of this paper is organized as follows. Section II provides background on caches, placement policies, and replacement policies. Section III describes the randomized replacement policy we explore here. Section IV presents our experimental methodology and empirical results evaluating randomized replacement in the context of caches with different placement policies. Section V discusses the possible impact of randomization on probabilistic WCET analysis. Section VI presents related work spanning cache architecture design, software randomization, and real-time systems. Finally, Section VII concludes with directions for future work.

## II. BACKGROUND

Caches are commonly used to hide the speed gap between the CPU and the main memory by exploiting

TABLE I  
POPULAR REPLACEMENT POLICIES AND HARDWARE PLATFORMS WHERE THEY HAVE BEEN IMPLEMENTED.

Replacement Policies	
Policy	Platforms
<i>Least Recently Used</i> (LRU)	Pentium I, MIPS 34K, LEON3
<i>First-In First-Out</i> (FIFO)	Xscale, ARM9, ARM11
<i>Most Recently Used</i> (MRU)	[6]
<i>Pseudo-LRU</i> (PLRU)	PowerPC 75x, Pentium II-IV

locality in memory accesses. In order to reduce the traffic overhead between the main memory and the cache, the memory space is logically split in memory blocks called *cache lines* (typically 32-128 bytes).

The behavior of a cache is determined by its **placement policy** and its **replacement policy**. A hash function is typically defined that indicates where in the cache a given object should be placed. These functions map the memory address to a *cache set* number. Each cache set is identified by certain bits of the memory address, called its *index*. Note that different cache lines can collide into the same cache set.

In order to avoid cache set conflicts, caches are partitioned into equally-sized cache sets, called *ways*. The size of each cache set is called the *k-associativity* of the cache. By doing this, different cache lines that collide into the same cache set can be distributed along the *k* different ways.

A *standard set-associative cache* (along called *standard cache*) is shown in Figure 1(a). Note that the hash function is the same for all the ways of the cache, so given an index, the same cache set is accessed in all cache ways.

Once all ways of a cache set are occupied, the *replacement policy* decides which cache line is evicted to make room for the new cache line. Numerous replacement policies have been proposed and implemented for both high performance and embedded processors; see Table II for an overview.

All these algorithms are based on deterministic information about the history of recent access patterns. Hence, when an access pattern is repeated, the outcome of these algorithms will be the same, which may potentially result in a *cache risk pattern* [7], i.e., a sequence of memory accesses that systematically evict cache lines that are still in use. A well-known example is a loop over a range of memory just larger than the cache, as shown in Figure 2.

### A. Skewed-Associative Caches

In order to reduce cache conflict misses, i.e., evicting cache-lines that are still in use, Bodin and Seznec

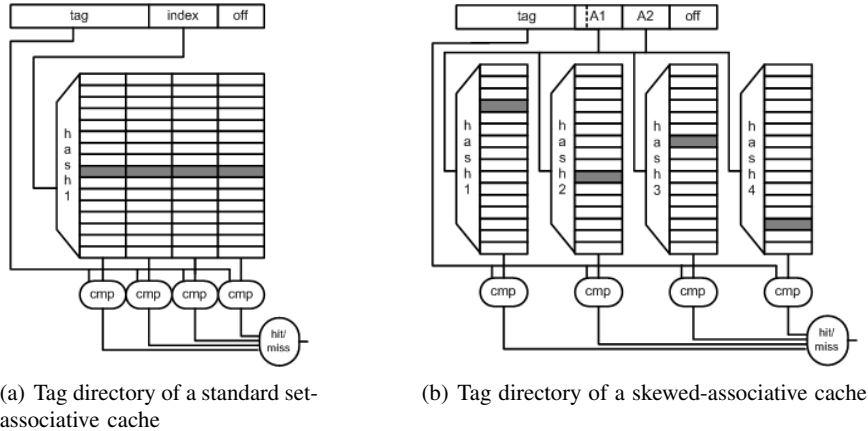


Fig. 1. Tag directory structures of two cache structures with different hash functions for mapping cache lines to ways.

```

for i=1 to 10
  a(); b(); c(); d(); e();
endfor

```

Fig. 2. Example code that causes pathological cache behavior for LRU. When all functions overlap on the same cache lines in a four-way set-associative cache, all function calls will trigger cache misses.

propose *skewed-associative caches*, or simply *skewed caches* [5]. Skewed caches use a placement policy that assigns a different hash function for each way of the cache, so that two addresses that collide in the same cache set in a given cache way are unlikely to also collide in the other ways (see Figure 1(b)). Skewed caches thus reduce the risk of colliding with a cache line that is still in use, because the placement policy achieves a better dispersion of data across the cache ways.

When using an LRU replacement policy (as recommended by Bodin and Sez nec), skewed caches can evict cache lines from a number of different cache sets, depending on where the least recently used cache line is located. Compare this to standard set-associative caches, where the evicted cache line always belongs to the same cache set, regardless of which way contains the least recently used cache line.

However, while skewed caches can increase hit rates over standard caches by reducing conflict misses, they suffer from the same worst-case behavior. Repeated execution of a pathological sequence that triggers catastrophic cache misses will continue to produce the same performance degradation.

### III. RANDOMIZED REPLACEMENT

The goal of using randomization in the context of cache replacement policies is to both reduce the likelihood of cache risk patterns and to virtually eliminate

TABLE II  
PERMUTATIONS OF PLACEMENT (LAYOUT) AND REPLACEMENT ALGORITHMS EXPLORED IN THIS PAPER.

Placement	Replacement	
	LRU	Random
Fixed	Standard [3]	<i>this paper</i>
Per Way	Skewed [5]	<i>this paper</i>

the risk of systematic performance degradation. Using randomization enables probabilistic analysis, allowing us to quantify the likelihood of a cache risk pattern as some fraction  $1/p$ . Because randomization makes each iteration effectively independent of the previous one, the odds of a cache risk pattern occurring in multiple iterations rapidly decreases: the likelihood that the risk pattern occurs in each of  $n$  iterations is just  $(1/p)^n$ .

We explore in this paper the effectiveness of randomized replacement policies, where the cache line to evict is always chosen at random instead of depending on recent access patterns (as with LRU). While evicting an object at random may seem certain to degrade performance, the worst-case effectiveness of some randomized eviction algorithms is better than LRU [8]. The reason for this is that it is unlikely that a random eviction algorithm will repeatedly evict a hot cache line, since the likelihood of evicting one particular cache line is  $1/k$ , where  $k$  is the associativity of the cache. Randomized replacement can degrade *average case* execution time, but because the focus of real-time systems is on reducing the *worst case* execution time, this reduction in performance is acceptable. Nonetheless, we show that this performance degradation is not significant and that random replacement in fact occasionally outperforms LRU-based policies.

Table III presents a grid describing the uses of randomized replacement we explore here. Because replacement policies are orthogonal to the placement (layout)

algorithms used, we can evaluate the effectiveness of randomized replacement both with standard set-associative caches and with skewed caches.

#### IV. THE EFFECT OF MEMORY LAYOUT

The cache replacement policy has a significant influence on the performance and the predictability of the cache. Deterministic replacement policies may result in cache risk patterns that systematically generate cache misses that, in turn, may potentially degrade the performance and increase the execution time variation. Since the cache access pattern is determined by the memory layout, the effectiveness of the replacement policy highly depends on the program memory layout, i.e., the set of memory addresses where the program is stored [4].

In this section, we quantify the cache performance impact of memory layouts by exploring all possible memory layouts of a simple fragment of code (see Figure 2) that resembles automatically-generated code (a common embedded-code structure), comparing random and LRU replacement policies. We use LRU because it is generally considered the best replacement policy in terms of performance and predictability [9], [7]. However, the same conclusions can be reached by using any of the deterministic replacement algorithms presented in Section II.

We demonstrate that, by using a random replacement policy, the likelihood of pathological performance is considerably reduced, improving the worst-case execution time and reducing the execution time variation of memory accesses.

##### A. Experimental Setup

All experiments presented in this section executed on an in-house cycle-accurate, execution-driven simulator compatible with Tricore ISA binaries [10]. The Tricore ISA, designed by Infineon Technologies, is widely used in hard real-time applications because it combines RISC, general-purpose and signal processing instructions within a single instruction set. The simulator was derived from CarCore [11]. We paid special attention to simulator correctness, extensively validating it through a wide range of tests.

The simulator models a memory hierarchy composed of a first level of cache (with separate instruction and data caches) and main memory. The first level of cache can model two different set-associativity caches placement policies: a standard set-associative cache and a skewed-associative cache. The former uses a fixed-hash function:  $\log_2(s)$  bits of the address, where  $s$  is the

TABLE III  
FIRST-LEVEL INSTRUCTION CACHE CONFIGURATION

Cache parameters	
Total Size	256 bytes
Cache line size	4 bytes
Associativity	4-way
Sets-per-way	16
Banks	1
Placement Policies	Fixed-Hash Function Skewed-Hash Function
Replacement Policies	LRU Random
Main Memory Access	30 cycles

number of cache sets, are used to index the cache. The latter uses a different hash function per way [5]:  $f_x(A) = s^x(A1) \otimes (A2)$ , where  $s^x(A1)$  is the  $x$ -bit circular shift of  $A1$  and  $A1$  and  $A2$  are the  $2 \times \log_2(s)$  address bits used to index the cache (see Figure 1(b)). Moreover, each type of cache can use two different replacement policies: random and LRU. Table III summarizes all cache models presented in this paper. Finally, the simulator models a main memory with a fixed latency of 30 processor cycles per access. The memory subsystem configuration used in the experiments is shown in Table IV-A.

In order to explore all possible memory layouts, we designed a small fragment of code that resembles automatically-generated code. The code, shown in Figure 2, is formed by five functions of similar size that are consecutively called inside a loop. The code is executed inside our processor model presented above, with a first level instruction cache of 256 bytes (see Table IV-A).

In order to consider conflict and capacity misses, we designed two different sample programs: one that fits inside the cache, with a total size of 132 bytes, and other that does not fit inside the cache, with a total size of 312 bytes. It is clear that while the former will suffer only conflict misses, the latter will suffer both capacity and conflict misses. In every new simulation, we initialize the state of the cache so in both cases, compulsory misses will also occur.

The reason for using such small and simple code is because it allows us to perform an exhaustive exploration of all possible memory layouts. Executing real applications in a more realistic cache configuration would result in a enormous space exploration making simulation and analysis infeasible. In fact, this is one of the reasons why predicting the cost of memory accesses in the presence of caches, both on average and in the worst-case, is exceedingly complex with current WCET analyses. In fact, although previous studies [7], [12] show that certain memory layouts result in different execution times of real

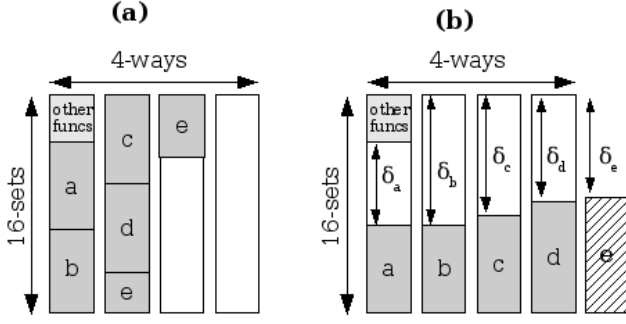


Fig. 3. Cache performance depends on memory layout: (a) Functions are placed consecutively in memory ( $\Delta = \langle 0, 0, 0, 0, 0 \rangle$ ), (b) Functions are not placed consecutively in memory ( $\Delta = \langle 24, 40, 40, 36, 32 \rangle$ ), resulting in a cache risk pattern.

applications, they do not conduct an exhaustive space exploration of the range of all possible memory layouts.

### B. Exploring All Memory Layouts

In order to explore all possible memory layouts, we assume that every function is separated in memory with respect to the previous function by  $\delta$  bytes. Therefore, considering the code presented in Figure 2, a memory layout can be represented by five  $\delta$  values, forming the  $\Delta$  vector:

$$\Delta = \langle \delta_a, \delta_b, \delta_c, \delta_d, \delta_e \rangle \quad (1)$$

Hence, every  $\delta_n$  shifts the beginning of the function  $n$  by  $\delta_n$  bytes, which has the same effect as adding padding code, resulting in a new cache pattern. The cache pattern sequences are a range of  $\delta$  values will repeat every  $\delta_n \bmod z$ , where  $z$  is the size in bytes of all cache lines that form a way. In our case, the cache pattern repeats every  $\delta_n \bmod 64$ .<sup>1</sup>

Figure 3 shows two different cache patterns (represented by its corresponding memory layout  $\Delta$  vectors) of the 132-byte code when executing in our 256-byte instruction cache. In Figure 3(a), the five functions are placed consecutively ( $\Delta = \langle 0, 0, 0, 0, 0 \rangle$ ) so they perfectly fit inside the cache. However, in Figure 3(b), the memory layout ( $\Delta = \langle 24, 40, 40, 36, 32 \rangle$ ) results in a pathological layout in which every function will systematically evict the next call function when using LRU, producing a chain of cache conflict misses.

By performing an exhaustive exploration of all possible memory layouts, we can quantify how every memory layout affects the execution time of memory accesses,

<sup>1</sup>Although the size of each way is 64 bytes, Tricore instructions have a length of 16 and 32 bits, so each  $\delta$  is required to be increased by 2 or 4 bytes.

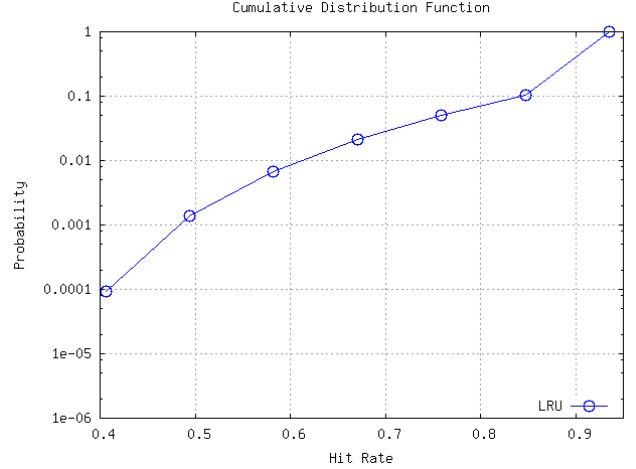


Fig. 4. Hit rate CDF of all possible memory layouts of a 132-byte code executed on a standard cache using LRU.

and so determine which memory layout results in pathological cache behavior. This analysis is illustrated in the following cases.

### C. Case Study: Code Fits Inside Cache

Figure 4 shows the Cumulative Distribution Function (CDF) of the hit ratios of all possible memory layouts, executing the 132-byte code in a standard cache that uses LRU. As expected, almost all memory layouts (about 90%) achieve the highest possible hit rate (0.94), which is the case when the layout yields no evictions. However, in the worst case, there is a set of memory layouts (approximately 1/10000 of all layouts) that achieve a very low hit rate (0.41). This high miss rate corresponds to a dramatic increase in execution time (see Section IV-F).

By randomizing the replacement policy, the decision to evict a cache set becomes independent of recent cache access pattern history. In order to quantitatively compare random and LRU replacement policies in a standard cache, we take one memory layout of every different hit rate obtained using LRU (represented by circles in Figure 4), and simulate  $10^6$  executions of each layout using LRU and random.

Figure 5 shows, from left to right, the CDF of the seven hit ratios (from 0.41 to 0.93) using LRU (labeled as *Standard & LRU*) and random (labeled as *Standard & Random*). As expected, the hit rate obtained using LRU in every execution is exactly the same because of its deterministic behavior.

However, when using the random replacement policy (*Standard & Random*), the memory layout that resulted in a cache risk pattern under LRU (first curve on the left) has considerably improved its hit rate ratio, obtaining

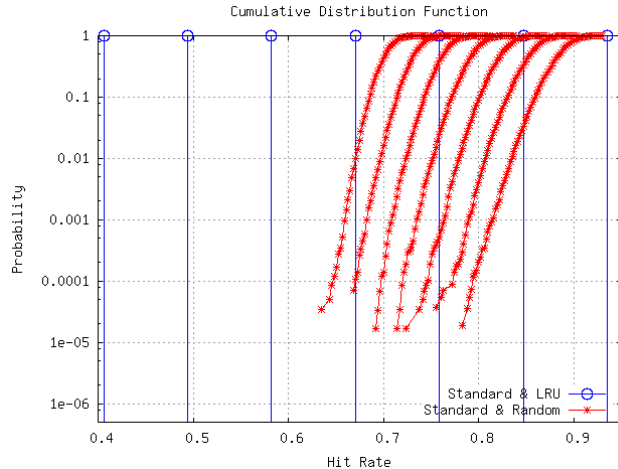


Fig. 5. From left to right, the curves show CDF of cache hit ratios of the 132-byte code executed into a standard cache using LRU (Standard & LRU) and random (Standard & Random) for the seven representative layouts.

0.64 in only a few executions. In fact, one can compute the probability of having a cache risk pattern using a random replacement policy by raising the probability of evicting a certain way to the power of the total amount of memory accesses. Since the variable that models the replacement policy is by design *independent and identically distributed*, the probability of evicting any way is  $1/k$ , where  $k$  is the associativity of the cache. In our case, the probability of pathological cache behavior using random replacement is extremely low:  $\sim 2.4^{-765}$ .

Moreover, randomized replacement also reduces the execution time variation of memory accesses. The worst-case layout yields a hit rate of 0.64, while the best-case yields a hit rate of 0.94.

#### D. Case Study: Code Does Not Fit Inside Cache

Figure 6 shows the CDF of the hit ratios of all possible memory layouts executing the 312-byte code in a standard cache that uses LRU. Since the code does not fit inside the cache, the cache suffers both capacity misses and conflict misses, resulting in very low hit rates. The hit rate varies from 0.41 (worst case) to 0.5 (best case). Note that this worst-case hit rate is the same as that obtained by the worst case for the 132-byte code (see Figure 4).

We then perform the same experiment presented in the previous section, with one memory layout of every hit rate obtained using LRU (represented by circles in Figure 6), and simulating  $10^6$  executions of each layout using LRU and random. Figure 7 shows, from left to right, the CDF of the six resultant hit-rates using LRU

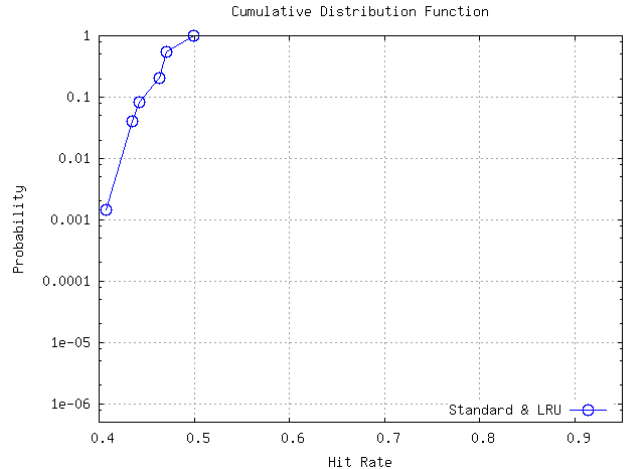


Fig. 6. Hit-rate CDF of all possible memory layouts of a 312-byte code executed into a standard cache using LRU.

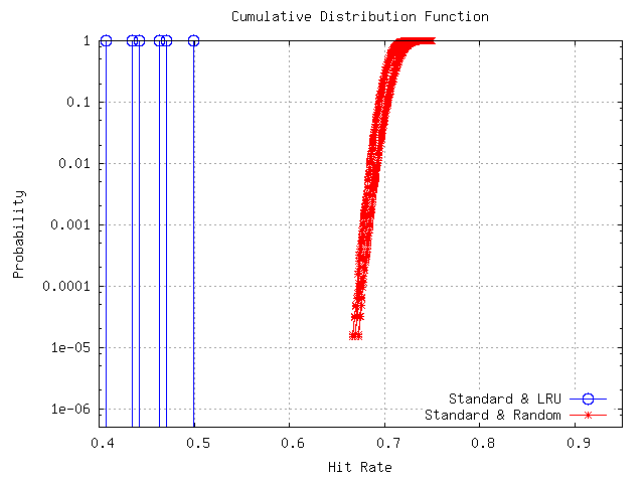


Fig. 7. From left to right, the curves show CDF of cache hit ratios of 312-byte code executed in a standard cache using LRU (Standard & LRU) and random (Standard & Random) for the six representative layouts.

(labeled as *Standard & LRU*) and random (labeled as *Standard & Random*).

The random replacement policy (*Standard & Random*) consistently improves the hit rate of the six representative layouts versus LRU. In addition, it virtually eliminates the variance between the worst and the best layouts, achieving a hit rate of 0.66 in the worst case and a hit rate of 0.74 in the best case for all memory layouts.

#### E. Skewed-Associate Caches

As explained in Section II-A, skewed caches can effectively increase the hit rate over those provided by standard caches, by applying different hash functions for each assigned way. Figure 8 shows the CDF of the hit ratios of all possible memory layouts when executing the

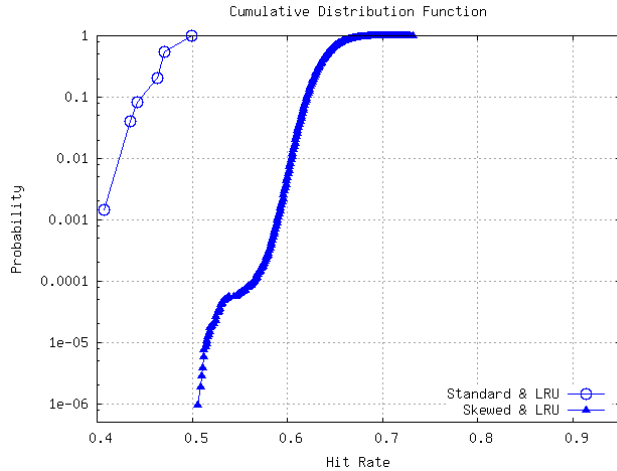


Fig. 8. Hit-rate CDF of all possible memory layouts of a 312-byte code when executing in a skewed cache (Standard & LRU) and a standard cache (Skewed & LRU), using LRU in both cases.

312-byte code in a standard cache (labeled as *Standard & LRU*) and a skewed cache (labeled as *Skewed & LRU*), applying in both cases LRU replacement policy.

The skewed cache consistently improves the performance of all memory layouts versus the standard cache, improving the hit rate of the standard cache by up to 0.62 and obtaining a maximum hit rate of 0.73. However, note that current skewed caches also suffer from pathological cases because of their deterministic behavior. Given a particular pathological access pattern, the skewed cache will always achieve the same (reduced) hit rate. Such pathological cases introduce a variation in the hit rate ratio from 0.73 to 0.51 when comparing the best and the worst layouts.

In order to see the effect of applying a randomized replacement policy to skewed caches, we take the best and the worst memory layouts obtained with the skewed cache using LRU, and then simulate  $10^6$  executions of each layout using both LRU and random. We also perform the same experiment with the layouts in a standard cache that uses random replacement.

Figure 9 shows, from left to right, the CDF of the hit-rate of the two layouts when executing them in a skewed-cache and LRU (labeled as *Skewed & LRU*), in a skewed-cache and random (labeled as *Skewed & Random*) and in a standard cache and random (labeled as *Standard & Random*).

By applying randomization, the worst and the best memory layouts achieve almost the same performance, varying the hit rate from 0.65 to 0.75. Note that the placement policy has no effect on performance, obtaining

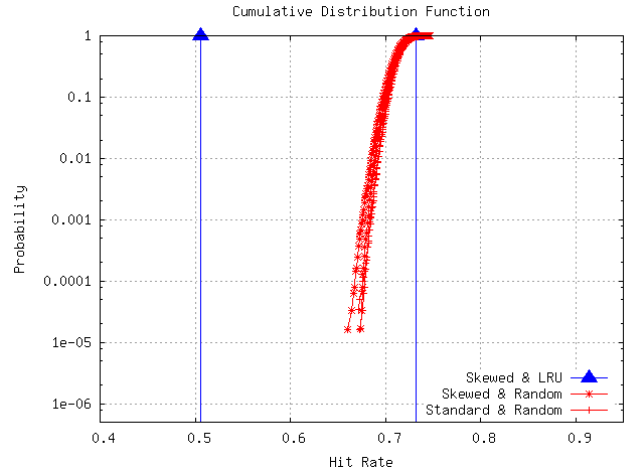


Fig. 9. From left to right, the curves show the CDF hit-rate of two representative memory layouts of the 312-byte code executed in a skewed-cache and LRU (*Skewed & LRU*) in a skewed-cache and random (*Skewed & Random*) and in a standard-cache and random (*Standard & Random*).

almost the same hit rate whether using a fixed-hash function or a skewed-hash function.

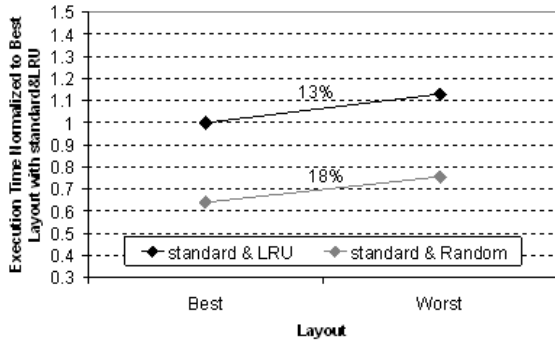
#### F. Execution Time Analysis

While increased cache misses generally lead to decreases in performance, the relationship is not linear. We compute the execution time obtained with the architecturally-detailed simulator presented in Section IV-A. In particular, we focus on the execution time of the best and worst case layouts of the 312-byte code for both standard and skewed caches with LRU and random replacement policies<sup>2</sup>. Recall that, in case of randomized replacement, these simulations comprise one million experiments.

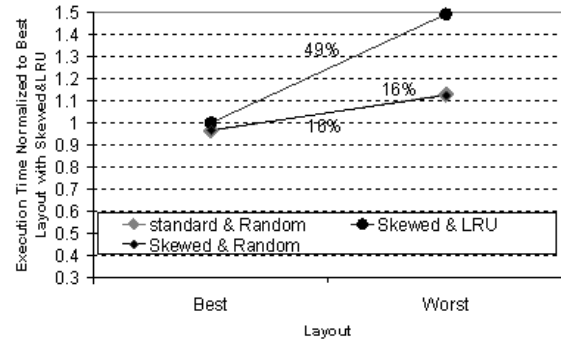
Figure 10 presents these results. Execution times are normalized to the execution time of the best memory layout obtained with a standard cache and LRU (Figure 10(a)) and with a skewed cache and LRU (Figure 10(b)). For randomized replacement with the best layout, we present the best execution time obtained along the one million executions. For the worst layout, we present the worst execution time obtained out of one million of executions. This approach allows us to compute the maximum execution time variation (shown on top of graph lines).

For standard caches, randomized replacement decreases runtime substantially both for the best layout

<sup>2</sup>We do not present the execution time analysis of the 132-byte code due to space limitations.



(a) Normalized execution times of the best and the worst layouts (obtained using a standard cache with LRU) of a standard cache with LRU and random.



(b) Normalized execution times of the best and the worst layouts (obtained using a skewed cache with LRU) of a skewed and standard caches with LRU and random.

Fig. 10. Normalized execution times and execution time variation of the 312-byte code executed in skewed and standard caches using LRU and random replacement policies.

(36% faster) and for the worst case (33% faster), although the execution time variation between the best and worst layout is slightly larger (from 13% to 18%).

For skewed caches, while randomized replacement achieves a smaller decrease for the best case (3%), it significantly improves runtime for the worst case (running almost 25% faster). Note that execution time is almost the same using a random replacement policy, whether with a standard cache or with a skewed cache. In effect, the placement policy has no effect on performance. Randomized replacement considerably decreases execution time variation versus LRU (from 49% to 16%).

## V. DISCUSSION

One key problem of WCET analysis in general is the issue of pessimism. In the case of the execution time analysis of memory accesses, when a particular access it is not possible to determine if it will lead to a cache hit or a cache miss, the only safe assumption is to consider that it will be a cache miss, i.e., the worst scenario. This kind of analysis may result in significant pessimism which limits their value since the chances that all these memory accesses result in cache misses is extremely small.

One way to address this issue is to move towards probabilistic timing analysis [13]. In this case, the goal is to compute the probability of the extreme case and then make an argument that the probability is sufficiently low. Unfortunately, current cache approaches are not amenable to such probabilistic arguments because the deterministic behavior of caches may result in pathological cases and systematic cache misses. Applying probabilistic arguments to current caches is unsafe precisely because probabilistic analysis techniques rely on a hypothesis of *statistical independence* [13]. Unless

one can prove that the variable that models the system or feature is independent and identically distributed (iid), using such statistical methods is not well-founded. Although several statistical approaches exist that deal with arbitrary models of dependence [14], they are not currently in widespread use.

However, applying random replacement policies instead of current deterministic approaches may allow us to argue that the execution time of memory accesses can be analyzed using probabilistic arguments, because the random variable modeling the system is independent and identically-distributed. Our hypothesis is that new advanced hardware features such as caches can be used and analysed effectively in hard real-time systems with designs that provide truly randomized behaviour. This shift will enable new probabilistic timing analysis techniques that can be used effectively in arguments of verification, demonstrating that the probability of pathological execution times is negligible.

## VI. RELATED WORK

### A. Randomized Architectures

In the early 90's, pseudo-random interleaved memory architectures were proposed to evenly distribute the sequence of references across the memory modules in order to achieve the full bandwidth of the memory system [15], [16].

Randomized caches were first proposed by Schlansker et al. [17] to eliminate the repetitive cache conflict misses caused by bad strides in high performance processors. They used a pseudo-random hash function to randomize addresses into cache sets. By doing this, they could develop a purely analytic approach to determine cache performance. A similar approach was proposed by

Topham et al. [18], in which a pseudo-random indexing scheme based on polynomial modulus functions were proposed, allowing eliminating bad strides inherent in some SPEC95 benchmarks.

To the best of our knowledge, randomization has never been proposed for safety-critical systems because of its inherent unpredictability. In fact, most of the proposals focus on static analysability. Moreover, recent studies focus on providing predictability to advance hardware features which allows using static analysis approaches [19], [20], [21]. These works provide deterministic behaviour to the different processor components, while we aim for truly randomized processor components.

### B. Replacement Policies for Real-Time

Reineke et al. present quantitative analytical results for the predictability of some replacement policies (LRU, FIFO, MRU and PLRU) based on two new metrics: *evict* and *fill* [9]. These metrics indicate how quickly the cache converges to a known state that can be statically predicted. The authors conclude that, based on these two metrics, the LRU replacement policy with an associativity up to eight performs better than others replacement policies. Recently, Junier et al. have presented interesting theoretical results that improve static instruction cache analysis methods for set-associative instruction caches with PLRU and a random replacement policies based on the *evict* and *fill* metrics [22]. As expected, the authors conclude that LRU replacement policy can be statically analyzed more tightly than PLRU and random replacement policies.

Bradford et al. compare twenty-one randomly chosen layouts of the same program obtaining an execution time variation of 22% [12]. More recently, Mezzetti et al. identify the problem of execution time variation caused by memory layouts in a real application: the on-Board Mission Time-line (MTL) component of the On-Board Software System used in the European Space Agency [7]. They develop three different memory layouts for the MTL and execute them on the AT697E LEON3 processor, obtaining an 11% execution time variation between a bad and a good memory layout, and argue that these underestimate the variance between the worst and the best memory layouts. The paper presents a list of cache design recommendations in which they recommend the use of LRU as a deterministic replacement policy.

### C. Layout Randomization

Randomized memory layouts were first used in the context of memory management in *DieHard*, proposed by Berger et al. [23]. *DieHard* places allocated heap objects randomly in a larger-than-required heap, enabling *probabilistic memory safety*, bounding the likelihood of memory errors such as buffer overflow, dangling pointers or invalid frees. As we observe here with randomized replacement, *DieHard*'s use of randomized placement generally does not significantly degrade application performance.

## VII. CONCLUSIONS AND FUTURE WORK

The features of modern microprocessors greatly complicate the prediction and analysis of worst-case execution times. In particular, this paper focuses on the complexities created by caches. Through execution-driven simulation and in the context of instruction caches, we show that standard cache replacement policies can lead to catastrophic cache behavior that increases cache misses to the point where it effectively disables the cache. This paper explores an alternative: the use of randomized replacement policies. We show that randomized replacement effectively avoids the pathological behaviors of deterministic replacement policies while achieving reasonable performance.

This paper represents our first step towards exploring the use of randomization as a way to prevent pathological cases and increase predictability in hard real-time systems.

We intend to extend this work in a number of directions. First, we plan to broaden the scope of our study to include larger benchmarks. Although this approach will preclude the possibility of exhaustive state space exploration, it will allow us to quantify the impact of randomized caches on execution time of larger applications.

Second, this paper limits its scope to the study of randomization in the replacement policy of a single level of the instruction cache. We plan to study the effects of randomization on worst-case performance for data caches and multiple levels of caches. We also plan to explore the possibility of randomized *placement* policies, where the mapping of a particular address to a cache line is not fixed.

Finally, we plan to develop the required analytical mathematical models to explore the benefits of randomization. These analytical models will primarily be obtained using standard probabilistic and statistical approaches such as extreme value statistics [24] or cop-

ulas [13]. This approach will allow deriving timing correctness from probabilistic guarantees. For example, if the requirements placed on the reliability of a sub-system indicate that the probability of a timing failure must be less than  $10^{-9}$  per hour of operation, then the analysis techniques developed will translate this reliability requirement into a probabilistic worst-case execution time for the sub-system. Moreover, we eventually intend to combine this work with an industrial-strength real-time WCET tool such as RapiTime to allow us to develop well-grounded probabilistic WCET estimates.

#### ACKNOWLEDGMENTS

This work has been supported by the Ministry of Science and Technology of Spain under contract TIN-2007-60625, the HiPEAC European Network of Excellence and by the MERASA STREP-FP7 European Project under the grant agreement number 216415. Emery Berger conducted this work as a visiting professor at the Barcelona Supercomputing Center, and is supported by the Barcelona Supercomputing Center, Intel, Microsoft Research, and the National Science Foundation under CAREER Award CNS-0347339 and CNS-0615211. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author(s) and do not necessarily reflect the views of the National Science Foundation. Authors would also like to thank Jörg Mische, Professor Theo Ungerer at University of Augsburg for their help in the integration of CarCore emulator into our simulation environment.

#### REFERENCES

- [1] A. Burns and A. Welling, *Real-Time Systems and Programming Languages*. Addison-Wesley, 1997.
- [2] R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström, “The worst-case execution-time problem—overview of methods and survey of tools,” *Trans. on Embedded Computing Sys.*, vol. 7, no. 3, pp. 1–53, 2008.
- [3] J. Hennessy and D. Patterson, *Computer Architecture, a Quantitative Approach*. Morgan Kaufmann, 2007, ch. 5. Memory Hierarchy Design, pp. 288–354.
- [4] G. Bernat, A. Colin, and J. Esteves, “Considerations on the LEON cache effects on the timing analysis of on-board applications,” in *DASIA 2007: Proceedings of the Data Systems In Aerospace Conference*, 2008.
- [5] F. Bodin and A. Sez nec, “Skewed associativity improves program performance and enhances predictability,” *IEEE Trans. Softw. Eng.*, vol. 23, no. 9, pp. 530–544, 1997.
- [6] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic, “Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite,” in *ACM-SE 42: Proceedings of the 42nd annual Southeast regional conference*. New York, NY, USA: ACM, 2004, pp. 267–272.
- [7] E. Mezzetti, N. Holsti, A. Colin, G. Bernat, and T. Vardanega, “Attacking the sources of unpredictability in the instruction cache behavior,” in *RTNS 2008: Proceedings of the 16th International Conference on Real-Time and Network Systems*, 2008.
- [8] A. Fiat, R. M. Karp, M. Luby, L. A. McGeoch, D. D. Sleator, and N. E. Young, “Competitive paging algorithms,” *J. Algorithms*, vol. 12, no. 4, pp. 685–699, 1991.
- [9] J. Reineke, D. Grund, C. Berg, and R. Wilhelm, “Timing predictability of cache replacement policies,” *Real-Time Syst.*, vol. 37, no. 2, pp. 99–122, 2007.
- [10] *Tricore 1. 32-bit Unified Processor Core v1.3*, Infineon, October 2005.
- [11] S. Uhrig, S. Maier, and T. Ungerer, “Toward a processor core for real-time capable autonomic systems,” in *Proc. Fifth IEEE International Symposium on Signal Processing and Information Technology*, 2005, pp. 19–22.
- [12] J. P. Bradford and R. W. Quong, “An empirical study on how program layout affects cache miss rates,” *ACM SIGMETRICS Performance Evaluation Review*, vol. 27, 1999.
- [13] G. Bernat, A. Burns, and M. Newby, “Probabilistic timing analysis: An approach using copulas,” *J. Embedded Comput.*, vol. 1, no. 2, pp. 179–194, 2005.
- [14] G. Bernat, A. Colin, and S. M. Petters, “WCET analysis of probabilistic hard real-time systems,” in *In Proceedings of the 23rd Real-Time Systems Symposium RTSS 2002*, 2002, pp. 279–288.
- [15] R. Raghavan and J. P. Hayes, “On randomly interleaved memories,” in *Supercomputing '90: Proceedings of the 1990 ACM/IEEE conference on Supercomputing*. Washington, DC, USA: IEEE Computer Society, 1990, pp. 49–58.
- [16] B. R. Rau, “Pseudo-randomly interleaved memory,” in *In Proceedings of the 18th Annual International Symposium on Computer Architecture*, 1991, pp. 74–83.
- [17] M. Schlansker, R. Shaw, and S. Sivaramakrishnan, “Randomization and associativity in the design of placement-insensitive caches,” *HP Report, HPL-93-41*, 1993.
- [18] N. Topham and A. González, “Randomized cache placement for eliminating conflicts,” *IEEE Trans. Comput.*, vol. 48, no. 2, pp. 185–192, 1999.
- [19] “<http://ginkgo.informatik.uni-augsburg.de/merasa-web>”
- [20] M. Paolieri, E. Quiones, F. Cazorla, G. Bernat, and M. Valero, “Hardware support for wcet analysis of hard real-time multicore systems,” in *ISCA '09: Proceedings of the 36th annual international symposium on Computer architecture*, 2009.
- [21] J. Rosen, A. Andrei, P. Eles, and Z. Peng, “Bus access optimization for predictable implementation of real-time applications on multiprocessor systems-on-chip,” in *Proc. 28th IEEE International Real-Time Systems Symposium RTSS 2007*, 2007, pp. 49–60.
- [22] A. Junier, D. Hardy, and I. Puaut, “Impact of instruction cache replacement policy on the tightness of wcet estimation,” in *Impact of instruction cache replacement policy on the tightness of WCET estimation*, Oct. 2008.
- [23] E. D. Berger and B. G. Zorn, “DieHard: Probabilistic memory safety for unsafe languages,” in *In Proceedings of the ACM SIGPLAN 2006 Conference on Programming Language Design and Implementation*. ACM Press, 2006, pp. 158–168.
- [24] K. Burry, *Statistical Methods in Applied Science*. John Wiley & Sons, 1975.