AVACS – Automatic Verification and Analysis of Complex Systems

REPORTS

of SFB/TR 14 AVACS

Editors: Board of SFB/TR 14 AVACS

Predictability of Cache Replacement Policies

by

Jan Reineke   Daniel Grund   Christoph Berg

Reinhard Wilhelm

AVACS Technical Report No. 9
September 2006
ISSN: 1860-9821

Publisher: Sonderforschungsbereich/Transregio 14 AVACS
(Automatic Verification and Analysis of Complex Systems)
Editors: Bernd Becker, Werner Damm, Martin Fränzle, Ernst-Rüdiger Olderog,
Andreas Podelski, Reinhard Wilhelm
ATRs (AVACS Technical Reports) are freely downloadable from `www.avacs.org`

Copyright © September 2006 by the author(s)
Author(s) contact: Reinhard Wilhelm (`rw@avacs.org`).

Predictability of Cache Replacement Policies*

Jan Reineke, Daniel Grund, Christoph Berg, and Reinhard Wilhelm

Universität des Saarlandes, Saarbrücken, Germany
{reineke,grund,cb,wilhelm}@cs.uni-sb.de

Abstract. Hard real-time systems must obey strict timing constraints. Therefore, one needs to derive guarantees on the worst-case execution times of the systems' tasks. In this context, predictable behavior of system components is crucial for the derivation of tight and thus useful bounds.

This paper presents results about the predictability of common cache replacement policies. To this end, we introduce two metrics that capture aspects of cache-state predictability. A thorough analysis of the LRU, FIFO, MRU, and PLRU policies yields the respective values under these metrics. To the best of our knowledge, this work presents the first quantitative, analytical results for the predictability of replacement policies. They support empirical evidence in static cache analysis.

1 Introduction

Embedded systems as they occur in application domains such as automotive, aeronautics, and industrial automation often have to satisfy hard real-time constraints. Timeliness of reactions is absolutely necessary. Off-line guarantees have to be derived using safe methods. Hardware architectures used in such systems now feature caches, deep pipelines, and all kinds of speculation to improve (average-case) performance. However, the same components are often disastrous to timing predictability. So, the system designer may find himself in the paradoxical situation that he has successfully raised the average-case performance of his system, but fails to derive sufficient timing guarantees despite his best efforts. This may be for two reasons: although the system's average-case behavior has improved, its worst-case performance has deteriorated. Even if the worst-case performance is sufficient, the provable bound may be too imprecise due to less predictable components. Hence, a system with good average-case, but with poor worst-case performance or low predictability will not be certifiable. [1] describes threats to the predictability of systems and proposes design principles that support timing predictability.

* This work has profited from discussions within the ARTIST2 Network of Excellence. It is supported by the German Research Foundation (DFG) as part of SFB/TR AVACS and by a scholarship in the GK 623.

1.1 Cache Analysis

Particular architectural components with a strong influence on both the average-case and the worst-case performance are the processor caches. Abstract interpretation has been successfully used to statically analyze the cache contents at program points [2]. Two static analyses, a *must-cache* and a *may-cache* analysis compute upper and lower approximations, resp., to the contents of all concrete caches that will occur whenever program execution reaches a program point. So, the *must-cache* at a program point says, what is definitely in each concrete cache whenever program execution reaches that program point. The *may-cache* says, what may be in a concrete cache whenever program execution reaches that program point. The result of the *must-cache* analysis is used to derive safe information about cache hits; the result of the *may-cache* analysis is used to safely predict information about cache misses.

Several properties of the processor caches influence predictability: associativity, replacement and write policy, and whether there are separated data and instruction caches [3]. Of these, the cache replacement policy has by far the strongest influence on the predictability of the cache behavior. We will investigate the following widely used replacement policies:

- Least Recently Used (LRU) used in Intel Pentium and MIPS 24K/34K
- First-In First-Out (FIFO), or Round-Robin, used in Intel XScale, ARM9, ARM11
- Most Recently Used (MRU) as described in [4,5]
- Pseudo-LRU (PLRU) used in PowerPC 75x and Intel Pentium II, III and IV

There are several reasons for the lack of information about cache contents. Static cache analyses usually have to begin with a completely unknown cache state because no information about the initial cache content is available. Control-flow joins, where analysis information needs to be safely combined, and statically unresolvable effective addresses can also introduce uncertainty.

Since information about the cache state may thus be unknown or lost, it is important to recover information quickly, to be able to classify memory accesses safely as cache hits or misses. The ability to recover such information greatly depends on the cache replacement policy employed.

1.2 Methodology

We show how quickly cache contents become known again when accessing a sequence of memory blocks starting from an unknown cache state. As updates of different sets are independent for the replacement policies we investigate, considering single cache sets is sufficient. We assume all memory accesses in the regarded sequences to be pairwise different. This is sensible because in most cases recurring accesses do not contribute additional information about the cache contents. For two replacement policies it is even impossible to ever recover full information about the cache state allowing *arbitrary* accesses.

We introduce two metrics parameterized with the associativity $k > 1$ that indicate how quickly cache state information can be regained.

evict(k): The number of distinct memory accesses to evict all non-accessed contents from an arbitrary cache set of size k. In other words, after *evict*(k) pairwise different memory accesses the cache set contains only elements from that access sequence.

fill(k): The number of distinct memory accesses to reach a completely known state, i.e. the cache contents and the future replacement behaviour are fully determined. For the considered replacement policies this means: From the point on when the cache set state is completely known, the cache set will always consist of the last k accessed elements. Figure 1 illustrates the two metrics.

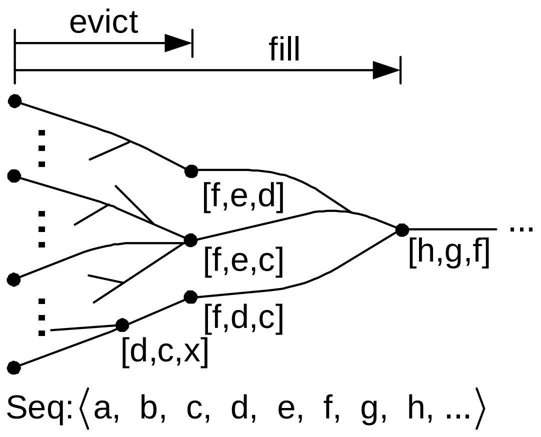


Fig. 1. Initially different cache sets converge, when accessing a sequence of distinct memory blocks. After *evict* accesses, any set contains only elements from the access sequence. *fill* accesses are required to fully converge to one known cache set. Note that several physical cache states may exhibit the same subsequent observable behavior. We do not distinguish such states.

evict(k) is interesting because it tells us at which point we can give safely predict that some elements are no more in the cache, i.e. the complement of *may*-information. Any element not contained in the last *evict*(k) accesses cannot be in the cache set. That is, the greater *evict*(k), the longer it takes to gain *less* precise *may*-information. After *fill*(k) distinct memory accesses we also know exactly what is contained in the cache, namely the last k accesses, i.e., we obtain *must*-information. At this point we can also give more precise answers to the *may*-

question: any element not contained in the last $k < evict(k)$ accesses cannot be in the cache set.

Consider the implications of these metrics on *any* cache analysis. They mark a limit on achievable precision: no analysis can infer any *may*-/precise *must*-information given an unknown cache state and less than $evict(k)/fill(k)$ distinct memory accesses. At the same time the metrics allow us to investigate the quality of different analyses. Does an analysis need longer access sequences to derive safe information about the cache contents, or is it optimal with respect to the metrics?

1.3 Notation

We distinguish between the kinds of accesses: if we assume all accesses are cache misses we denote this with the subscript M, otherwise we will use HM. Finally, the cache replacement policy is indicated in the superscript. Thus $fill_{HM}^{LRU}(8)$ is the number of distinct accesses (hits or misses) needed to know the exact contents of an 8-way cache set using LRU replacement. For brevity, we will also use $e(k)$ and $f(k)$ for $evict(k)$ and $fill(k)$.

We will denote the state of a cache set in the form $[a, b, c, d]$, with $a, \ldots, d$ the lines in the set. $\perp$ denotes invalid lines. Memory access sequences are given in the form $\langle b, c, d\rangle$. Sequences can be concatenated: $\langle b, c, d\rangle \circ \langle f, e\rangle = \langle b, c, d, f, e\rangle$.

2 LRU Caches

LRU replacement maintains a queue of length k, where k is the associativity of the cache. If an element is accessed that is not yet in the cache (a miss), it is placed at the front of the queue. The last element of the queue is then removed. It is the least recently used element of those in the queue. At a cache hit, the element is moved from its position in the queue to the front, effectively treating hits and misses equally.

The contents of LRU caches are very easy to predict. Having only distinct accesses and a strict least recently used replacement, directly yields the tight bounds

$$e_{HM}^{LRU}(k) = e_{M}^{LRU}(k) = f_{HM}^{LRU}(k) = f_{M}^{LRU}(k) = k.$$

3 FIFO Caches

FIFO can also be seen as a queue: new elements are inserted at the front evicting elements at the end of the queue. In contrast to LRU, hits do not change the queue. Implementations use a round-robin replacement counter for each set pointing to the cache line to replace next. This counter is increased if an element is inserted into a set, while a hit does not change this counter.

In the case of misses only, FIFO behaves like LRU. Thus, the following tight bounds are obvious:

$$e_M^{FIFO}(k) = f_M^{FIFO}(k) = k.$$

Lemma 1 (Surviving Elements). *Of $i \leq 2k-1$ pairwise different accesses, at least $\lceil \frac{i}{2} \rceil$ survive in a FIFO cache set.*

Proof. Assume there were m misses and h hits, $m+h=i$. If $m \geq h$, every miss places an element at the front of the queue, and the number of known elements is $\min(m,k) \geq \lceil \frac{i}{2} \rceil$, even if all elements that had a hit are evicted.

If $m \leq h$, we use the fact that each miss evicts at most one 'known' element from the cache, while inserting itself. Hence, with $h \leq k$ the number of known elements in the cache is at least $m + (h-m) = h \geq \lceil \frac{i}{2} \rceil$.

Lemma 2 ($evict_{HM}^{FIFO}$). *After accessing at most $2k-1$ pairwise distinct elements in a k-way FIFO cache, the cache contains only elements from these $2k-1$ accesses. This bound is tight.*

Proof. Using Lemma 1 with $i = 2k-1$ gives $e_{HM}^{FIFO}(k) \leq 2k-1$. The following example shows the tightness. The access sequence $\langle x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-1} \rangle$ of length $2k-2$ conducted on the initial cache state $[z, x_1, \ldots, x_{k-1}]$ results in the state $[y_{k-1}, \ldots, y_1, z]$. Since z survived, $e_{HM}^{FIFO}(k) > 2k-2$.

Lemma 3 ($fill_{HM}^{FIFO}$). *One needs at most $3k-1$ accesses for any initial cache state to reach a completely known cache state. This bound is tight.*

Proof. After $2k-1$ accesses, no more hits can occur, due to Lemma 2. Since the next k accesses will be misses, $3k-1$ is a bound on $f_{HM}^{FIFO}(k)$. It is also a tight bound as shown by a similar example as in the proof of Lemma 2. Again, assume initial cache state $[z, x_1, \ldots, x_{k-1}]$. The sequence $\langle x_1, \ldots, x_{k-1} \rangle \circ \langle y_1, \ldots, y_{k-1} \rangle \circ \langle z, w_1, \ldots, w_{k-1} \rangle$ of length $3k-2$ results in the cache state $[w_{k-1}, \ldots, w_1, y_{k-1}]$, which does not contain z, i.e. $f_{HM}^{FIFO}(k) > 3k-2$.

4 MRU Caches

MRU stores one status bit for each cache line. In the following, we call these bits MRU-bits. Every access to a line sets its MRU-bit to 1, indicating that the line was recently used. Whenever the last 0 bit of a set is set to 1, all other bits are reset to 0. At cache misses, the line with lowest index (in our representation the left-most) whose MRU-bit is 0 is replaced.

We represent the state of an MRU cache set as $[a,b,c,d]_{0101}$, where 0101 are the MRU-bits and $a, \ldots, d$ are the contents of the set. On this state an access to e yields a cache miss and new state $[e,b,c,d]_{1101}$. Accessing d leaves the state unchanged. A hit on c forces a reset of the MRU-bits: $[e,b,c,d]_{0010}$.

Lemma 4 ($evict_M^{MRU}$ and $evict_{HM}^{MRU}$).

$$evict_M^{MRU}(k) = evict_{HM}^{MRU}(k) = 2k-2$$

gives a tight bound on the number of misses/accesses sufficient to evict all entries from a k-way set-associative MRU cache set.

Proof. We prove the tight bounds by showing $2k-2$ to be an upper bound for $evict_{HM}^{MRU}$ and a lower bound for $evict_{M}^{MRU}$. This suffices to prove the tightness for both, since by definition $evict_M \leq evict_{HM}$.

For the lower bound, consider the initial cache state $s = [x_1, \ldots, x_k]_{0\ldots001}$ and access sequence $\langle y_1, \ldots, y_{k-1}\rangle \circ \langle z_1, \ldots, z_{k-2}\rangle$. After the first part, the MRU-bits are reset and state $s' = [y_1, \ldots, y_{k-1}, x_k]_{0\ldots010}$ results. The second part of the sequence replaces $y_1, \ldots, y_{k-2}$ resulting in state $s'' = [z_1, \ldots, z_{k-2}, y_{k-1}, x_k]_{1\ldots110}$. x_k is still part of the set proving $evict_{M}^{MRU}(k) > 2k-3$.

For the upper bound, notice that at some point during any k distinct accesses (hits or misses), the MRU-bits are reset. MRU-bits of lines that have not been accessed until this point are then set to 0. If it took k accesses to reset the bits, exactly these k elements make up the cache set. Otherwise (less than k accesses), after the reset $k-1$ MRU-bits are 0 and an additional $k-1$ accesses are sufficient because accesses to elements with MRU-bit 1 are impossible, from the reset point on. They would be hits and violate the property of distinct accesses.

Lemma 5 **($fill^{MRU}$).** *For the MRU replacement policy it is impossible to give a bound on the number of accesses needed to reach a completely known cache state:*

$$fill_{HM}^{MRU}(k) = fill_{M}^{MRU}(k) = \infty$$

Proof. Consider an access sequence of distinct accesses. After at most $2k-2$ accesses there will be only misses. Thus a cache state $s = [x_1, \ldots, x_k]_{0\ldots01}$ will eventually occur for some $x_1, \ldots, x_k$. It will take $2k-2$ further misses to eliminate x_k, hence future states following s will not consist of the last k accessed elements. Even worse, we will reach similar states $[y_1, \ldots, y_k]_{0\ldots01}$ over and over again.

The next two lemmas compensate this gap in the results by giving results similar to $fill^{MRU}(k)$.

Lemma 6. *Consider an MRU cache state $[x_1, \ldots, x_k]_{0\ldots010\ldots0}$ and an access sequence that only produces misses. Every element from that sequence will remain in the cache for at least $k-1$ accesses.*

Proof. Consider an arbitrary element e of the sequence. Since elements are inserted from left to right, all elements in the set left of e will be replaced earlier (after the next reset). Right to e there can be at most one element with MRU-bit 1. Thus, at least $k-2$ other lines will be accessed before the next reset and thus before e is replaced.

Lemma 7. *Let $k > 2$. After at most $2k-4$ misses the last $k-1$ accessed elements are present in the cache set, and the set is stable with respect to this weaker property. This bound is tight.*

Proof. The first reset of the MRU-bits occurs after at most $k-1$ accesses. If it takes exactly $k-1$ accesses the initial cache state fits the requirements of Lemma 6 proving the lemma for this case. Otherwise, the reset takes place after at most $k-2$ accesses. $k-2$ additional accesses are sufficient due to Lemma 6 because the miss causing the reset has an MRU-bit of 1 and cannot be evicted by the next $k-2$ misses.

Tightness is shown by the initial state $[x_1, \ldots, x_k]_{0\ldots011}$ and the sequence $\langle y_1, \ldots, y_{k-2}\rangle \circ \langle z_1, \ldots, z_{k-3}\rangle$: $[x_1, \ldots, x_k]_{0\ldots011} \rightarrow [y_1, \ldots, y_{k-2}, x_{k-1}, x_k]_{0\ldots0100}$ $\rightarrow [z_1, \ldots, z_{k-3}, y_{k-2}, x_{k-1}, x_k]_{1\ldots100}$. $y_{k-2}, z_1, \ldots, z_{k-3}$ are the last $k-2$ misses but neither x_{k-1} nor x_k which are still in the cache belong to the last $k-1$ misses.

Lemma 8. *Let $k > 2$. After at most $3k-4$ accesses (hits or misses) the last $k-1$ accessed elements are present in the cache set and the set is stable with respect to this weaker property. This bound is tight.*

Proof. Due to our general assumption about distinct accesses it holds that after the MRU-bits have been reset the second time, no more hits are possible because every line has been accessed at least once: every MRU-bit must have been 0 at some time and 1 later on. Now, Lemma 6 is applicable and $k-2$ further accesses are sufficient.

The first reset occurs after at most k accesses, the second one after exactly $k-1$ additional accesses. Adding the $k-2$ accesses after the second reset yields $3k-3$. We now exclude the cases where k accesses are needed for the first reset proving the upper bound of $3k-4$: if exactly k accesses were needed to reset the bits for the first time every cache line with MRU-bit 1 must have been accessed. Thus there are no further hits possible after the first reset, already.

Consider the following cache states and access sequences:
$[x_1, \ldots, x_{k-1}, x_k]_{0\ldots00011}$
$\rightarrow \langle x_k, u_1, \ldots, u_{k-2}\rangle$
$\rightarrow [u_1, \ldots, u_{k-2}, x_{k-1}, x_k]_{0\ldots00100}$
$\rightarrow \langle v_1, \ldots, v_{k-4}, x_{k-1}\rangle$
$\rightarrow [v_1, \ldots, v_{k-4}, u_{k-3}, u_{k-2}, x_{k-1}, x_k]_{1\ldots10110}$
$\rightarrow \langle v_{k-3}, v_{k-2}\rangle$
$\rightarrow [v_1, \ldots, v_{k-4}, v_{k-3}, u_{k-2}, x_{k-1}, v_{k-2}]_{0\ldots00001}$
$\rightarrow \langle w_1, \ldots, w_{k-3}\rangle$
$\rightarrow [w_1, \ldots, w_{k-3}, u_{k-2}, x_{k-1}, v_{k-2}]_{1\ldots11001}$.

The last $k-1$ accesses were $v_{k-3}, v_{k-2}, w_1, \ldots, w_{k-3}$, but v_{k-3} has just been evicted by w_{k-3}. Only the next access (evicting u_{k-2}) will make sure the last $k-1$ accessed elements are present in the cache set.

This shows the tightness for $k > 2$. Note that for $k = 4$ the accesses $v_1, \ldots, v_{k-4}$ and the MRU-bit prefixes $0\ldots0$ and $1\ldots1$ do not exist.

5 PLRU Caches

PLRU (Pseudo-LRU) is a tree-based approximation of the LRU policy. It arranges ways in a tree with $k-1$ "tree bits" pointing to the line to be replaced next. A 0 indicates the left subtree, a 1 the right. See Figure 2 for an explanation of the replacement policy. PLRU is much cheaper to implement than true LRU in terms of storage requirements and update logic. This comes at a price: it does not always replace the least recently used element, which reduces predictability.

PLRU also tracks invalid lines. On a cache miss, invalid lines are filled from left to right, ignoring the tree bits. The tree bits are still updated.

Since illustrating the states of these cache sets is rather complicated we introduce the notion of a *normalized cache state*. With no invalid lines, equivalent cache states with same content and same order of replacements can be obtained by interchanging neighboring subtrees and flipping the corresponding tree bit. To represent a concrete cache set we choose an equivalent one with all tree bits set to 1. For instance the concrete cache state $[a, b, c, d]_{010}$ with tree bits 010 in Figure 2 is represented by $[d, c, a, b]^{\cong}$. Disregarding invalid lines the right-most element will be replaced in the normalized representation on a cache miss; it is pointed at by the tree bits. An access moves an element to the left-most position.

An *access path* to a cache line is a sequence of bits indicating the directions one has to take to walk from the root to this line in the normalized representation of the cache set; 0 for left, 1 for right. E.g. the access path of d in $[a, b, c, d, e, f, g, h]^{\cong}$ is 011.

We will interpret access paths as binary numbers. We will use two operators: $\overleftarrow{p_1 \dots p_n} = p_n \dots p_1$ to reverse the order of bits and $\overline{1100101} = 0011010$ to invert bits on paths.

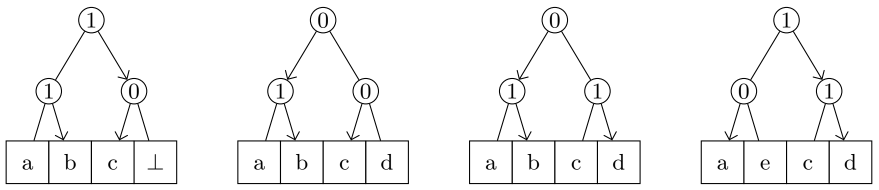
Observation 1 (Access Path Update) *Consider elements $a \neq b$ with access paths p_a and p_b. Let $p_a = pre \circ p_1 \circ post_a$ and $p_b = pre \circ \overline{p_1} \circ post_b$, i.e. p_a and p_b have a common (possibly empty) prefix until they diverge and finish with (possibly empty) suffixes $post_a$ and $post_b$, respectively. Accessing b moves it to the front with access path $p'_b = 0 \dots 0$. Since a and b share a prefix, flipping the bits on the path to b also affects a's prefix: its new access path is $0 \dots 01 \circ post_a$.*

Lemma 9 (Replacement Distance). *A cache line with access path $p_1 \dots p_n$ will be replaced after $\overline{p_n \dots p_1} + 1$ consecutive misses assuming no invalid lines.*

Proof. Assuming no invalid lines, all misses will go to access path $1 \dots 1$. Each miss decrements $\overline{p_n \dots p_1}$ by 1 for $p_1 \dots p_n \neq 1 \dots 1$: consider the dissection of $p_1 \dots p_n$ into $1 \dots 10 p_{post}$. A miss updates $p_1 \dots p_n$ to $0 \dots 01 p_{post}$ by Observation 1. For $\overline{p_n \dots p_1}$ this means going from $\overline{\overleftarrow{p_{post}}} 10 \dots 0$ to $\overline{\overleftarrow{p_{post}}} 01 \dots 1$.

The cache line d with access path 011 from the example above will be replaced after $001 + 1 = 2$ consecutive misses: $011 \rightarrow 111 \rightarrow$ replaced.

Lemma 10 (Minimal Replacement Distance). *After accessing an element, it takes at least $log_2 k + 1$ further accesses to evict that element from the set. In other words, the last $log_2 k + 1$ accesses to a cache set always reside in the set.*



(a) Initial cache set state $[a, b, c, \bot]_{110}$ with representation $[a, b, \bot, c]^{\cong}$.

(b) After a miss on d it becomes $[d, c, a, b]^{\cong}$.

(c) After a hit on c it becomes $[c, d, a, b]^{\cong}$.

(d) After a miss on e it becomes $[e, a, c, d]^{\cong}$.

Fig. 2. Three accesses to a 4-way associative PLRU cache set: a miss on d followed by a hit on c and a miss on e. On a miss, one allocates invalid lines from left to right. If all lines are valid one replaces the line the tree bits point to. After every access all tree bits on the path from the accessed line to the root are set to point away from the line. Other tree bits are left untouched.

Proof. After the access to an element its access path is $0 \ldots 0$. To replace this element all bits on its access path must be flipped to $1 \ldots 1$. By Observation 1 each access to other elements flips at most one of the bits of the access path to 1. To reach the lower bound of $log_2 k + 1$ one must access the neighboring subtrees in a bottom-up fashion, to avoid flipping bits back to 0.

5.1 Eviction

Lemma 11 ($evict_M^{PLRU}$)**.**

$$evict_M^{PLRU}(k) = \begin{cases} 2k - \sqrt{2k} : k = 2^{2i+1}, i \in \mathbb{N}_0 \\ 2k - \frac{3}{2}\sqrt{k} : otherwise \end{cases}$$

is a tight bound on the number of misses to evict all entries from a k-way set-associative PLRU cache set.

Proof. Assuming no invalid lines, this problem is an easy one. k misses suffice to evict a complete set. If all lines are invalid, the problem is equally easy. It becomes more complicated if some subset of size $0 < i < k$ of the lines is invalid. The first i misses will then go into these invalid lines, instead of following the standard PLRU replacement policy. These accesses do however modify the tree bits in the standard way, as if they had been hits.

The number of misses needed to completely evict the cache set is then determined by the positions of the remaining $k' = k - i$ non-accessed lines. Each line can be associated with the number of misses necessary to replace the content of that line. By Lemma 9 the line with access path $p_1 \ldots p_n$ will be replaced after $\overline{p_n \ldots p_1} + 1$ consecutive misses, i.e. the number of trailing 0s in $p_1 \ldots p_n$ mainly determines the replacement distance. To have m trailing 0s none of the

$2^m - 1$ neighbors in the particular subtree of height m may have been accessed in the first phase, filling up the invalid lines. Any access in the subtree would have flipped at least one of the final m bits. If k' lines have not been accessed yet, the maximal number of trailing 0s in any of these lines' access paths may be $\lfloor log_2 k' \rfloor$.

So, the maximal distance to eviction of any untouched line is bounded by

$$\overline{\underbrace{0\ldots0}_{\lfloor log_2 k' \rfloor}10\ldots0} + 1 = \underbrace{1\ldots1}_{\lfloor log_2 k' \rfloor}01\ldots1 + 1$$

$$= \underbrace{1\ldots1}_{\lfloor log_2 k' \rfloor + 1}0\ldots0 = \underbrace{1\ldots1}_{log_2 k} - \underbrace{1\ldots1}_{log_2 k - (\lfloor log_2 k' \rfloor + 1)}$$

$$= (2^{log_2 k} - 1) - (2^{log_2 k - (\lfloor log_2 k' \rfloor + 1)} - 1) = k - \frac{k}{2^{\lfloor log_2 k' \rfloor + 1}}$$

All in all, we get $z = i + k - \frac{k}{2^{\lfloor log_2 k' \rfloor + 1}} = 2k - k' - \frac{k}{2^{\lfloor log_2 k' \rfloor + 1}}$ as an upper bound for the number of accesses to evict a PLRU-set with misses only. Obviously, z is maximized by a power of two (for any non power of two $k' = 2^l + \delta, 0 < \delta < 2^l$, $k'' = 2^l$ results in a higher value of z), which allows us to simplify the formula to $2k - k' - \frac{k}{2k'}$, assuming k' is a power of two. Maximizing this yields

$$evict_M^{PLRU}(k) = \begin{cases} 2k - \sqrt{2k} & : k = 2^{2i+1}, i \in \mathbb{N}_0 \\ 2k - \frac{3}{2}\sqrt{k} & : \text{otherwise} \end{cases}$$

with

$$k' = \begin{cases} \frac{1}{2}\sqrt{2k} & : k = 2^{2i+1}, i \in \mathbb{N}_0 \\ \sqrt{k} & : \text{otherwise} \end{cases}$$

This proves the given $evict_M^{PLRU}(k)$ to be an upper bound. To prove its tightness we can give access sequences and initial cache configurations that exactly reach the bounds. Assume $[\bot_1, \ldots, \bot_{k-k'}, x_1, \ldots x_{k'}]$ with arbitrary tree bits as the initial configuration. Then, the access sequence $\langle y_1, \ldots, y_{k-k'} \rangle$ results in the normalized cache state $\left[y_{i_1}, \ldots, y_{i_{k'}}, x_{i_1}, \ldots, x_{i_{k'}}, y_{i_{k'+1}}, \ldots, y_{i_{k-k'}}\right]^{\cong}$. Observe that the access path $0\ldots01\underbrace{0\ldots0}_{log_2 k'}$ leads to x_{i_1}. By Lemma 9, it takes $\underbrace{1\ldots1}_{log_2 k'}01\ldots1 + 1 = k - \frac{k}{2k'}$ further misses to eliminate x_{i_1}. Together with the $k - k'$ previous accesses to fill the invalid lines, it sums up to the given upper bound, proving its tightness.

If one cannot assume that only misses will occur, the number of accesses for eviction gets even larger. However, we do not have to consider invalid lines because allocations to invalid lines are equivalent to hits at those position.

Lemma 12. *The number of accesses to the two halves c_1, c_2 of a $2k$-way cache set differs by at most k.*

Proof. Consider a situation with h_i hits and m_i misses to c_i. For each but the first miss on c_2 there must be an access to c_1 to flip the bits back to c_2: $h_1+m_1 \geq m_2-1$. Thus the difference $d = h_2+m_2-(h_1+m_1) \leq m_2+h_2-m_2+1 = h_2+1$. If $h_2 < k$ then $d \leq k$. The last possible case is $h_2 = k$, in which all hits h_2 must have preceded all misses m_2. But every further access to c_2 must then be directly preceded by at least one access to c_1 again yielding $d \leq k$.

Lemma 13 **(**$evict_{HM}^{PLRU}$**).** *It takes at most* $\frac{k}{2}\log_2 k+1$ *pairwise distinct accesses to evict all entries from a* k*-way set-associative PLRU cache set. Again, this is a tight bound.*

Proof. Claim: let $z(k)$ be an upper bound for the number accesses needed for a cache set of associativity k. Then $z(2k) = 2z(k) + k - 1$ is an upper bound for a set of associativity $2k$.

We consider a set of size $2k$ to be composed of two halves c_1, c_2 of size k. Wlog. let c_1 be the first half with no initial contents left. Let a_1 and a_2 be the number of accesses on c_1 and c_2 respectively to reach this state. Then c_2 needs at most $z(k) - a_2$ further accesses. Since c_1 consists of elements from the access sequence only, every subsequent access to c_1 will be a miss. Therefore, there can be at most one access to c_1 between two consecutive accesses to c_2 from now on.

Combining the last two statements there can be at most $2(z(k) - a_2) - 1$ further accesses until c_2 is completed, too. Adding the first $a_1 + a_2$ accesses results in $a_1 + a_2 + 2(z(k) - a_2) - 1 = 2z(k) + a_1 - a_2 - 1$. Using Lemma 12 we can bound $z(2k)$ by $2z(k) + k - 1$.

Solving the recurrence for z with the trivial value $z(2) = 2$ proves the theorem.

To prove tightness assume a worst-case initial cache state c_k and a worst case access sequence $s_k = \langle u_1, \ldots, u_{z(k)}\rangle$ for associativity k are known. The access sequence $\langle x_1, \ldots, x_k, u_1, v_1, \ldots, u_{z(k)-1}, v_{z(k)-1}, u_{z(k)}\rangle$ evicts the cache with initial state $[x_1, \ldots, x_k] \circ c_k$ with no less than $k + 2z(k) - 1$ accesses.

For $k = 2$ all cache states and all access sequences of length 2 are worst case initial cache states serving as a basis for the recursion.

5.2 Fill

Lemma 14 **(**$fill_{M}^{PLRU}$**).** *After at most* $fill_{M}^{PLRU}(k) = 2k - 1$ *misses the cache state is completely known. Tight for* $k > 2$*. For* $k = 2$*, 2 is an obvious tight bound.*

Proof. At most k misses can go into invalid lines. The last of these accesses resides in the line with access path $0\ldots0$ in the normalized cache. According to Lemma 9, it will be evicted after k further misses, i.e. the $k-1$ subsequent misses fill up the cache. Further misses result in a FIFO behavior. The following example proves tightness: assume the initial cache state $c = [\perp_1, \ldots, \perp_k]$ consisting of invalid lines only. Now, consider the access sequence $\langle x_1, \ldots, x_k\rangle \circ \langle y_1, \ldots, y_{k-2}\rangle$. After processing $\langle x_1, \ldots, x_{\frac{k}{2}}\rangle$ $x_{\frac{k}{2}}$ has access $0\ldots0$. The next accesses x_i go to the other half of c. Thus, the access paths of $x_{\frac{k}{2}}$ and x_i have no common prefix. By Observation 1, $x_{\frac{k}{2}}$ has access path $10\ldots0$ after $\langle x_{\frac{k}{2}+1},\ldots,x_k\rangle$. By Lemma 9, it will take $1\ldots10+1 = k-1$ further misses to eliminate it, after $k-2$ accesses it is still in the cache. Thus, the cache does not consist of the last k accessed elements, in particular it has not stabilized yet.

Lemma 15. *If it takes $evict_{HM}^{PLRU}(k)$ accesses to evict a cache set, the last two accesses must have gone to different halves of the cache set.*

Proof. Assuming this is false one could insert an additional miss-access between the last two accesses on the half not accessed. Thus the number of accesses for eviction would be increased by one contradicting the assumption of a worst case.

Lemma 16 **($fill_{HM}^{PLRU}$).** *After at most $\frac{k}{2}\log_2 k+k-1$ pairwise different accesses the PLRU cache state is completely known. This bound is tight.*

Proof. We want to prove the given bound based on our results for $evict_{HM}^{PLRU}(k)$. The difference $fill_{HM}^{PLRU}(k) - evict_{HM}^{PLRU}(k)$ is $k-2$. Since the last access to a set always resides in the left-most position with access path $0\ldots0$, $k-1$ additional misses suffice to fill the set due to Lemma 9. This still leaves us one short of the given bound if eviction took exactly $evict_{HM}^{PLRU}(k)$ steps. In that case, however, the last two accesses must have gone to different halves due to Lemma 15. Thus, they have access paths $0\ldots0$ and $10\ldots0$. Due to Lemma 9 they will be replaced after k and $k-1$ misses. Thus $k-2$ further accesses suffice.

Tightness is shown by modifying a generic worst-case example for $e_{HM}^{PLRU}(k)$. Let $s = \langle x_1,\ldots,x_e\rangle$ be this worst-case access sequence (assuming the same initial cache state). Then $s' = \langle x_1,\ldots,x_{e-2},h\rangle\circ\langle y_1,\ldots,y_{k-2}\rangle$ of length $evict_{HM}^{PLRU}(k)+k-3$ results in the intermediate cache state $[h,\ldots,x_{e-2},\ldots|x_{e-3},\ldots]^{\cong}$ where $|$ denotes the center of the cache set. The final state is $\left[y_{i_1},\ldots,y_{i_{k-2}},x_{e-3}\right]^{\cong}$.

Effectively, we remove the last two accesses from the old example and insert a hit h into the access sequence accessing the left side of the (normalized) cache. Knowing that the last two accesses x_{e-3}, x_{e-2} accessed different halves of the set[1], the hit h changes the order in which these two elements will be replaced. Thus x_{e-3} must be evicted from the set to stabilize it. Due to Lemma 9 this takes $k-1$ additional accesses because x_{e-3} has access path $10\ldots0$ after the hit. Carrying out s' only, will result in the cache state depicted above, which is not yet stabilized.

6 Related Work

Sleator and Tarjan [6] consider replacement policies from a different point of view. They investigate the amortized efficiency of list update and paging rules. Comparing the online paging rules LRU, FIFO, LIFO, and LFU to Belady's optimal policy OPT, they show that in the worst case any online algorithm

[1] this is due to the construction of our former worst-case example, cf. Lemma 13

must fare worse than OPT by a certain factor and go on to prove that LRU and FIFO do perform as well as possible for an online algorithm. This work concerns performance rather than predictability of replacement policies.

Al-Zoubi et al. [4] perform several measurements using the SPEC CPU2000 benchmarks. They compare the performance of different associativities and replacement policies including FIFO, LRU, PLRU, MRU, and OPT. They conclude that LRU, PLRU, and MRU show nearly the same performance. These policies are approximately as good as a cache of half the size with OPT policy while clearly outperforming FIFO. This interesting experimental result yields new insights concerning performance in practice. It does however, not deal with predictability.

In [3] Heckmann et al. provide *must* and *may* analyses for LRU, PLRU and a pseudo round-robin replacement policy in the context of worst-case execution time tools. Cache lines are assigned ages where "old" lines are close to eviction. Newly introduced lines assume the minimum age 0. Updates change these ages to account for all possible concrete scenarios: in the *may* analysis, the minimal possible age is taken, in the *must* analysis the maximal. For LRU, this yields very precise and efficient analyses. For PLRU, the *must* analysis loses precision while staying efficient, it can maximally infer 4 of the 8 lines of a 8-way set-associative PLRU cache which is strongly related to our Lemma 10. The *may* analysis becomes useless since only ages 0 and 1 are reachable. They also give an example for a replacement policy with very poor predictability: pseudo round-robin used in the Motorola ColdFire 5307. It is effectively a FIFO replacement except that the replacement counter is shared among *all* cache sets. The inability to analyze the sets independently results in an even lower predictability than for the FIFO policy.

The manual of the PowerPC 75x series [7] gives the number of uniquely addressed misses to flush an 8-way PLRU cache set used in these CPUs, which is an instance of $evict_M^{PLRU}$.

7 Conclusions and Future Work

An important part in the design of hard real-time systems is the proof of punctuality which is determined by the worst-case performance of the system. Performance boosting components like caches have an increasing impact on both the average and the worst-case performance. We investigated the predictability of four prominent cache replacement policies. We introduced the metrics *evict* and $fill$ and derived exact bounds for them.

In these metrics, no policy can perform better than LRU because k is an obvious lower bound for any replacement policy. The other policies under investigation, PLRU, MRU, and FIFO, perform considerably worse: in the more interesting cases of $e_{HM}(k)$ and $f_{HM}(k)$, FIFO and MRU exhibit linear growth in terms of k, where PLRU grows super-linearly. However, instantiating k with the common values 4 and 8 shows a different picture. Here, PLRU even fares slightly better than FIFO and MRU. Yet, compared to 8-way LRU, PLRU, MRU,

Table 1. Summary of the main results for all policies.

Policy	$e_M(k)$	$f_M(k)$	$e_{HM}(k)$	$f_{HM}(k)$
LRU	k	k	k	k
FIFO	k	k	$2k-1$	$3k-1$
MRU	$2k-2$	$\infty/2k-4^{\dagger}$	$2k-2$	$\infty/3k-4^{\dagger}$
PLRU	$\left\{\begin{matrix} 2k-\sqrt{2k} \\ 2k-\frac{3}{2}\sqrt{k} \end{matrix}\right\}$	$2k-1$	$\frac{k}{2}\log_2 k+1$	$\frac{k}{2}\log_2 k+k-1$

Table 2. Examples for $evict$ and $fill$ for $k = 4, 8$.

	$k=4$				$k=8$			
Policy	e_M	f_M	e_{HM}	f_{HM}	e_M	f_M	e_{HM}	f_{HM}
LRU	4	4	4	4	8	8	8	8
FIFO	4	4	7	11	8	8	15	23
MRU	6	$\infty/4$	6	$\infty/8$	14	$\infty/12$	14	$\infty/20$
PLRU	5	7	5	7	12	15	13	19

and FIFO take more than twice as long to regain full information. In particular, this differs from the worst-case *performance* results obtained in [6], where FIFO and LRU fared equally well. Our results support previous practical experience in static cache analysis [3].

Future work could drop the restriction that all elements of access sequences are different. This would allow for the construction of precise and efficient (as possible) cache analyses. A first step would be to investigate the normalization of arbitrary access sequences, e.g. $\langle x_1, \ldots, x_n, y, y\rangle$ can be simplified to $\langle x_1, \ldots, x_n, y\rangle$ in all replacement policies we considered. For LRU it suffices to keep the last access to each element within the sequence, which means keeping at most k elements. Can we do something similar regarding FIFO or PLRU?

The metrics allow us to investigate the precision of different analyses. Does an analysis ever regain full *may*/*must*-information? If so, does it need longer access sequences to derive safe information about the cache contents than suggested by $fill(k)$ and $evict(k)$, or is it optimal with respect to these metrics?

References

1. L. Thiele and R. Wilhelm, "Design for timing predictability." *Real-Time Systems*, vol. 28, no. 2-3, pp. 157–177, 2004.
2. C. Ferdinand and R. Wilhelm, "Efficient and precise cache behavior prediction for real-time systems." *Real-Time Systems*, vol. 17, no. 2-3, pp. 131–181, 1999.
3. R. Heckmann, M. Langenbach, S. Thesing, and R. Wilhelm, "The influence of processor architecture on the design and the results of WCET tools," *Proceedings of the IEEE*, vol. 91, no. 7, pp. 1038–1054, 2003.

† See Lemma 7 and Lemma 8.

4. 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 Press, 2004, pp. 267–272.
5. A. Malamy, R. Patel, and N. Hayes, “Methods and apparatus for implementing a pseudo-lru cache memory replacement scheme with a locking feature,” United States Patent 5029072, October 1994.
6. D. D. Sleator and R. E. Tarjan, “Amortized efficiency of list update and paging rules,” *Commun. ACM*, vol. 28, no. 2, pp. 202–208, 1985.
7. Freescale Semiconductor Inc., “MPC750 RISC microprocessor user manual, section 3.5.1,” http://www.freescale.com/files/32bit/doc/ref_manual/MPC750UM.pdf, Freescale Semiconductor, Inc., 1 2002.