Real-Time Syst (2007) 37: 99–122
DOI 10.1007/s11241-007-9032-3

Timing predictability of cache replacement policies

Jan Reineke · Daniel Grund · Christoph Berg · Reinhard Wilhelm

Published online: 23 August 2007

Abstract Hard real-time systems must obey strict timing constraints. Therefore, one needs to derive guarantees on the worst-case execution times of a system's 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 three metrics, *evict*, *fill*, and *mls* 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. Our results support empirical evidence in static cache analysis.

Keywords Predictability · Timing analysis · Cache analysis · Cache replacement policies · Hard real-time systems

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 on the worst-case

J. Reineke (✉) · D. Grund · C. Berg · R. Wilhelm
Universität des Saarlandes, Saarbrücken, Germany
e-mail: reineke@cs.uni-sb.de

D. Grund
e-mail: grund@cs.uni-sb.de

C. Berg
e-mail: cb@cs.uni-sb.de

R. Wilhelm
e-mail: wilhelm@cs.uni-sb.de

execution time of each task have to be derived using safe methods. Execution times of a task vary depending on the task's inputs and the initial hardware state. The vast number of cases prohibits exhaustive testing to exactly determine the worst-case execution time. Instead approximative methods have to be applied. Such methods must be conservative, i.e., they must never underestimate the worst-case execution time, they must statically overapproximate the dynamic behavior of a task on all possible inputs and hardware states.

Caches, deep pipelines, and all kinds of speculation are increasingly used in todays embedded systems to improve average-case performance. At the same time they increase the variability of execution times of instructions due to the possibility of timing accidents with high penalties: a cache miss may take 100 times as long as a cache hit. Thus, the precision (tightness of upper bounds) of a static analysis greatly depends on its ability to statically exclude as much detrimental behavior to the timing of the program's instructions as possible: cache misses, mispredicted branches, pipeline stalls, etc. Exclusion of these so-called timing accidents tightens the upper bound by the associated timing penalty, e.g., the cache miss penalty or the time to refill the pipeline. Examples of static analyses to exclude timing accidents can be found in Langenbach et al. (2002), Thesing (2004), Ferdinand and Wilhelm (1999).

A designer that introduces caches, deep pipelines, or other performance boosting components into his system may find himself in the paradoxical situation that he has successfully improved the average-case performance of the 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 low predictability of the new components. Hence, a system with good average-case, but with poor worst-case performance or low predictability will not be certifiable. Thiele and Wilhelm (2004) describes threats to the predictability of systems and proposes design principles that support timing predictability.

The timing predictability of a system is a measure for the possibility of determining tight bounds on execution times. As depicted in Fig. 1, timing predictability is composed of uncertainty and associated penalties. Uncertainty comprises timing accidents that cannot be excluded statically but never happen during execution. High penalties do not automatically make a system unpredictable: if there is no uncertainty

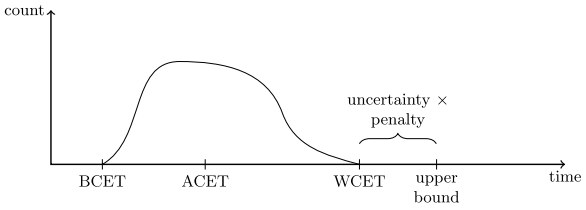


Fig. 1 Execution times of tasks vary depending on inputs and the initial state of the hardware they are executed on. The figure depicts a distribution of execution times. The border cases are known as Best- and Worst-Case Execution Time (BCET and WCET). A correct timing analysis obtains a safe upper bound on all possible execution times

this is not a problem. On the other hand, high levels of uncertainty only become harmful to timing predictability if the associated penalties are large.

Caches As noted before, the processor caches have a strong influence on both the average-case and the worst-case performance. Due to the high cache-miss penalties they have a potentially strong impact on the predictability of a system. Several properties of the processor caches influence predictability: associativity, replacement policy, write policy, and whether there are separated data and instruction caches, see Heckmann et al. (2003). 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 regarding their timing predictability:

- Least Recently Used (LRU) used in Intel Pentium I 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 Al-Zoubi et al. (2004), Malamy et al. (1994)
- Pseudo-LRU (PLRU) used in PowerPC 75x and Intel Pentium II-IV

The cache miss penalty is the same for all of the considered replacement policies. Timing predictability of cache replacement policies therefore only depends on the amount of uncertainty.

1.1 Contributions

We introduce two metrics, *evict* and *fill*, that capture our notion of the predictability of cache replacement policies.

Every cache analysis has to cope with a certain amount of uncertainty resulting from various sources explained in Sect. 3.1. The two metrics, *evict* and *fill* indicate how quickly knowledge about cache hits and misses can be (re-)obtained. They mark a limit on the precision that *any* cache analysis can achieve. A thorough analysis of the LRU, FIFO, MRU, and PLRU policies yields the respective values under these metrics.

Further analyses elaborate on these results and yield a more refined view on the limits of cache analyses: While *evict* and *fill* constitute milestones in the recovery of information, supplementary results show how information evolves in between.

To the best of our knowledge, this work presents the first quantitative, analytical results about the predictability of replacement policies.

2 Caches

Caches are commonly employed to hide the speed gap between main memory and the CPU by exploiting locality in memory accesses. They are very fast but small memories that store a subset of the main memory's contents. On today's architectures a cache miss may have an associated penalty of several hundred CPU cycles. Future architectures are expected to exhibit even larger cache miss penalties.

To reduce traffic and management overhead, the main memory is logically partitioned into *memory blocks* of size B bytes. Memory blocks are cached as a whole in

cache lines of equal size. Usually, B is a power of two. This way the block number is determined by the most significant bits of a memory address.

When accessing a memory block one has to determine whether the memory block is stored in the cache (cache hit) or not (cache miss). To enable an efficient look-up, each memory block can be stored in a small number of cache lines only. 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. Again, k is usually a power of two, such that the set number is determined by the least significant bits of the block number. The remaining bits, known as the *tag* are stored along with the data to finally decide, whether and where a memory block is cached within a set.

Since the number of memory blocks that map to a set is far greater than the associativity of the cache, a so-called replacement policy must decide which memory block to replace upon a cache miss. To facilitate useful replacement decisions a number of status bits is maintained that store information about previous accesses. We only consider replacement policies that have independent status bits per cache set. Almost all known policies comply with this.

3 Cache analysis

In cache analysis there is a concept of *may* and *must* cache information at program points: *may*- and *must*-caches are upper and lower approximations, respectively, 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 is a set of memory blocks that are definitely in each concrete cache at that point. The *may*-cache is a set of memory blocks that may be in a concrete cache whenever program execution reaches that program point. *May* and *must* cache information is obtained by static analysis. Ferdinand and Wilhelm (1999), for instance, presents cache analyses based on abstract interpretation.

Must cache information is used to derive safe information about cache hits; in other words it is used to exclude the timing accident "cache miss". The complement of the *may* cache information is used to safely predict cache misses. The more cache hits can be predicted, the better the upper bound on the worst-case execution time will be. Vice versa, predicting more cache misses will result in a better lower bound on the best-case execution time. Observe the asymmetry between *may*- and *must*: while a greater *must*-cache means more precise information, a greater *may*-cache means less precise information.

3.1 Sources of uncertainty

There are several reasons for uncertainty about cache contents:

- Static cache analyses usually cannot make any assumptions about the initial cache contents. Cache contents on entrance depend on previously executed tasks. Even assuming a completely empty cache may not be conservative as shown in Thesing (2004). The only safe initial *must*-cache is the empty set, whereas the only safe initial *may*-cache must contain every memory block that may be mapped to the particular cache set.

- At control-flow joins, analysis information about different paths needs to be safely combined. Intuitively, one must take the intersection of the incoming *must*-information and the union of the incoming *may*-information. A memory block can only be in the *must*-cache if it is in the *must*-caches of all predecessor control-flow nodes, correspondingly for *may*-caches.
- If the analysis cannot exactly determine the address of a memory access it must conservatively account for all possible addresses. This especially deteriorates *may*-information.
- Statically undetermined preempting tasks may change the cache state at preemption points.

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. Fortunately, this is possible for most caches. The speed of this recovery greatly depends on the cache replacement policy employed and influences uncertainty about cache hits and misses. Thus, it is an indicator of timing predictability.

4 Cache predictability metrics

For a timing analysis the data that is actually cached is irrelevant. Only the address ranges that are cached influence timing. In the following, if we talk about *cache contents*, we only really talk about the addresses of the cached memory blocks. We show how quickly cache contents become known when accessing a sequence of memory blocks starting from an unknown cache state. For the replacement policies we consider, an access to a cache set does not affect the state of other sets. Thus, we consider the recovery of information about single cache sets.

4.1 Notation and basic notions

We use the following generic names and notations:

$a, b, c \in A$	the set of memory addresses
$[b, e, c, f], q \in C^{p_k}$	the set of cache-set states of associativity k under policy p
$\langle b, c, d \rangle, s \in S \subseteq A^*$	the set of access sequences with pairwise different accesses
$\circ : S \times S \to S$	concatenation of two sequences
$CC^{p_k} : C^{p_k} \cup S \to 2^A$	the set of memory addresses of memory blocks of a cache-set state or of an access sequence
$update^{p_k} : C^{p_k} \times S \to C^{p_k}$	cache-set state after accessing a sequence under policy p

Individual cache-set states are denoted by $[b, e, c, f]$. Depending on the replacement policy additional status bits as e.g. in $[e, b, c, d]_{0010}$ are used to fully describe a state. Their meaning will become clear in the description of the particular policy.

We assume all memory accesses in the regarded sequences to be *pairwise different*. This is sensible because recurring accesses do not contribute additional information about the cache contents. Another reason is that *arbitrarily* long access sequences can be constructed for two of the considered replacement policies, namely PLRU and MRU, that never recover complete information about the cache contents if repetitive accesses are allowed. In other words, there are access sequences such that different initial states result in different states for an arbitrary number of accesses; they never converge.

May- and must-information available after observing an access sequence s without knowing the initial set state can be defined as follows:

$$May^{p_k}(s) := \bigcup_{q \in C^{p_k}} CC^{p_k}(update^{p_k}(q, s)),$$

$$Must^{p_k}(s) := \bigcap_{q \in C^{p_k}} CC^{p_k}(update^{p_k}(q, s)).$$

$May^{p_k}(s)$ is the set of cache contents that may still be in the cache set after accessing the sequence s, regardless of the initial cache state. Analogously, $Must^{p_k}(s)$ is the set of cache contents that must be in the cache set after accessing the sequence s. Since we take into account every initial state, $Must^{p_k}(s)$ is always a subset of $CC^{p_k}(s)$.

The following two definitions show *how much* may- and must-information is available after observing any access sequence s of length n:

$$may^{p_k}(n) := |May^{p_k}(s)|, \quad \text{where } s \in S, |s| = n,$$

$$must^{p_k}(n) := |Must^{p_k}(s)|, \quad \text{where } s \in S, |s| = n.$$

Note that $may^{p_k}(n)$ and $must^{p_k}(n)$ are well-defined: For all sequences s of length n, $|May^{p_k}(s)|$ is equal (the same goes for $|Must^{p_k}(s)|$). The sequences contain pairwise different accesses only and are thus equal up to renaming. Thus, $May^{p_k}(s_1)$ equals $May^{p_k}(s_2)$ up to renaming, too. In the following proofs we may therefore always restrict our attention to one representative access sequence.

4.2 Metrics

Based on $may^{p_k}(n)$ and $must^{p_k}(n)$ we are ready to define *evict* and *fill* that indicate how quickly may- and must-information can be recovered:

$$evict^p(k) := \min\{n : may^{p_k}(n) \leq n\},$$

$$fill^p(k) := \min\{n : must^{p_k}(n) = k\}.$$

Figure 2 illustrates the two metrics. $evict^p(k)$ tells us at which point we can safely predict that some elements are no more in the cache, i.e. they are in the complement of may-information. Any element not contained in the last $evict^p(k)$ accesses cannot be in the cache set: If some element not contained in the sequence could have

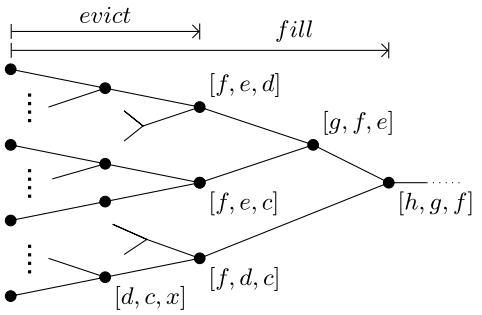


Fig. 2 Initially different cache sets converge when accessing a sequence $\langle a, b, c, d, e, f, g, h, \ldots \rangle$ of pairwise different memory blocks. After *evict* accesses, any set contains only elements from the access sequence. *fill* accesses are required to converge to one completely known cache set. Selected cache sets are annotated with their respective contents

"survived" then any other element not contained in the sequence could have "survived" as well. Then $may^{p_k}(n) = c > n$ where c is the number of blocks that map to the cache set. Less than $evict^p(k)$ accesses do not allow to predict any misses. The greater $evict^p(k)$, the longer it takes to gain may-information, and furthermore, the obtained may-information is *less* precise. The obtained may-information is less precise, because any of the greater number of $evict^p(k)$ elements may still be in the cache set.

After $fill^p(k)$ pairwise different memory accesses we know exactly what is contained in the cache set, namely the last k accesses, i.e., we obtain complete may- and must-information. This allows us to precisely predict cache hits and misses. In contrast to may-information, some must-information is directly obtained with the first memory accesses. At least the most recently accessed element is in the cache set. Thus, it is pointless to define a counterpart to *evict* for must-information, since $\min\{n \mid must^{p_k}(n) \geq 1\} = 1$ for all policies.

Consider the implications of these metrics on *any* cache analysis. They mark a limit on achievable precision: no analysis can infer any may-information (complete must-information) given an unknown cache-set state and less than $evict(k)$ ($fill(k)$) pairwise different 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 set contents, or is it optimal with respect to the metrics?

Another application of these metrics is to determine the minimal effort to establish a desired cache-set state, assuming that no explicit instructions are available to do so. This may be used to eliminate initial uncertainty in cache analyses by prepending load instructions. Or simply to create uniform conditions for performance measurements. For this special purpose, it is interesting to investigate access sequences resulting in cache misses only. In such a case, a desired cache-set state can be obtained faster. We therefore distinguish M- and HM-access sequences: if we assume all accesses in the regarded sequences to be cache misses we denote this by the subscript M, otherwise by HM. Thus $fill^{\mathrm{LRU}}_{\mathrm{HM}}(8)$ is the number of pairwise different 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 $fill(k)$.

As we have noted above, *some* must-information can be immediately obtained with one individual access. The following definition of the *minimal life-span* captures

how this generalizes:

$$mls^p(k) := \max\{n \ : \ must^{p_k}(n) = n\}.$$

The minimal life-span is the minimal number of accesses necessary to evict an element out of a cache set that has just been accessed (not counting the access that possibly brought the element into the set). In other words, it tells us how many of the most-recently accessed elements are always in the cache.

Based on the minimal life-span of a policy, it is easy to determine some must-information: the last $mls(k)$ accessed elements are always in the cache set. Therefore, one can construct a must-analysis that remembers the last $mls(k)$ accesses. A high value of $mls(k)$ would make this a reasonably precise must-analysis. Depending on the replacement policy, optimal analyses that eventually obtain complete must-information may be much more expensive.

4.3 Equalities

The definitions given in the previous subsection were chosen to be as uniform as possible: they all relate $must^{p_k}(n)$ and $may^{p_k}(n)$ with k and n. However, for the following proofs we need to establish some equalities to ease argumentation about *evict*, *fill*, and *mls*.

Lemma 1 *$evict^p(k)$ is the minimal length of access sequences such that only elements of the sequence may be contained in the cache set.*

$$evict^p(k) = \min\{n \mid \forall s \in S, |s| = n : May^{p_k}(s) \subseteq CC^{p_k}(s)\}.$$

Proof We need to show $may^{p_k}(n) \leq n \Leftrightarrow \forall s \in S, |s| = n : May^{p_k}(s) \subseteq CC^{p_k}(s)$.

$\Leftarrow$ is clear since $|CC^{p_k}(s)| = |s| = n$ and therefore $|May^{p_k}(s)| \leq n$.

$\Rightarrow$: Assume $May^{p_k}(s) \not\subseteq CC^{p_k}(s)$ for some s. Then at least one element a not contained in s must have survived. Upon an access, the update process of the status bits is independent of the tag bits of all non-accessed elements. Thus, the tag bits of a can be chosen arbitrarily. I.e. any other element $b \notin CC^{p_k}(s)$ could have survived as well. Then, $may^{p_k}(n) = c > n$ where c is the number of blocks that map to the cache set. □

Lemma 2 *This following equation makes explicit that the cache set is filled with the last k accesses of the access sequence s, once its state is known.*

$$fill^p(k) = \min\{n \mid \forall s \in S, |s| = n, s = s_1 \circ s_2, |s_2| = k : Must^{p_k}(s) = CC^{p_k}(s_2)\}.$$

Proof One needs to show $must^{p_k}(n) = k \Leftrightarrow \forall s \in S, |s| = n, s = s_1 \circ s_2, |s_2| = k : Must^{p_k}(s) = CC^{p_k}(s_2)$.

The $\Leftarrow$ direction of the equivalence is obvious. For $\Rightarrow$ one needs to show that whenever $must^{p_k}(n) = k$ then for any sequence s of length $n \geq k$, $Must^{p_k}(s) = CC^{p_k}(s_2)$. This holds because $CC^{p_k}(s_2) \supseteq Must^{p_k}(s)$: For any sequence there is an initial state, such that the sequence will result in misses only. Therefore, one

of the intersected sets is always equal to $CC^{p_k}(s_2)$. In addition $|CC^{p_k}(s_2)| = k$. As $|Must^{p_k}(s)| = k$, $CC^{p_k}(s_2)$ and $Must^{p_k}(s)$ must be equal. □

Lemma 3 *An address a that has just been accessed will at least remain in the cache set for the $mls^p(k)$ subsequent accesses.*

$$mls^p(k) = \max\{n \mid \forall s \in S, |s| \leq n : a \in Must_k(a \circ s)\}.$$

Proof We need to show $must^{p_k}(n) = n \Leftrightarrow \forall s \in S, |s| \leq n : a \in Must_k(a \circ s)$.

$\Rightarrow$: $\forall s : Must^{p_k}(s) \subset CC^{p_k}(s)$. Since $must^{p_k}(n) = n$ all accessed elements, including the first must be in $Must^{p_k}(s)$.

$\Leftarrow$: $must^{p_k}(n) \geq n$ since the last n elements are always contained in the cache set. Obviously $must^{p_k}(n) \leq n$. □

5 LRU caches

LRU replacement conceptually maintains a queue of length k for each cache set, 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 if the set is full. 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, in this respect treating hits and misses equally.

The contents of LRU caches are very easy to predict. For memory access sequences with pairwise different accesses and a strict least-recently-used replacement, we obtain the tight bounds

$$evict_{\mathrm{HM}}^{\mathrm{LRU}}(k) = evict_{\mathrm{M}}^{\mathrm{LRU}}(k) = fill_{\mathrm{HM}}^{\mathrm{LRU}}(k) = fill_{\mathrm{M}}^{\mathrm{LRU}}(k) = mls^{\mathrm{LRU}}(k) = k.$$

$evict(k)$ and $fill(k)$ tell us at which point any may- and complete must-information can be determined. However, the metrics do not tell us how may- and must-information evolves before and after these points. For the common case of an 8-way associative cache, we have precisely determined how much may- and must-information is available as a function in the number of accesses. Note that these functions mark the maximum information that can be obtained; a particular analysis may be less precise. Figure 3 shows plots of these functions. In the case of LRU replacement these functions are quite obvious, which will be different in the following cases of FIFO, MRU, and PLRU. Must-information rises with every access up to the minimal life-span $mls^{\mathrm{LRU}}(8)$, which is equal to $fill_{\mathrm{HM}}^{\mathrm{LRU}}(8)$ and $evict_{\mathrm{HM}}^{\mathrm{LRU}}(8)$. Up to $evict(k)$ accesses, any memory block mapped to the cache set may reside in the set.

We have determined these functions by exhaustively generating all successor states of all possible initial cache-set states, exploiting symmetries. For LRU and FIFO replacement this could have been done analytically, but for the other cases this would have been very tedious. This automatic computation was only possible up to associativity 8 as the number of states grows rapidly with rising associativity.

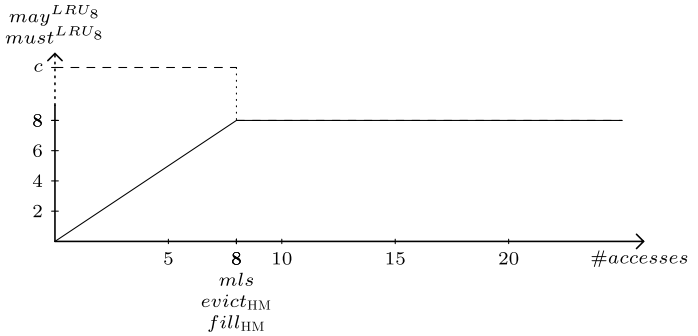


Fig. 3 Evolution of may- and must-information of a 8-way LRU cache set. c is the number of blocks that can be mapped to the cache set. May- and must-information is shown by the *dashed* and the *solid curve*, respectively. From $fill(k)$ on the two functions have the same value

6 FIFO caches

FIFO cache sets 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. Our representation of FIFO cache sets has to be interpreted in this way: In $[b, c, e, d]$, d will be replaced on a miss resulting in $[x, b, c, e]$.

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:

$$evict_{\mathrm{M}}^{\mathrm{FIFO}}(k) = fill_{\mathrm{M}}^{\mathrm{FIFO}}(k) = k.$$

Lemma 4 (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, $m+h=i$. First, assume $m \geq h$. Every miss places an element at the front of the queue, and the number of known elements is $\min(m, k) \geq \lceil \frac{i}{2} \rceil$.

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

Theorem 1 ($evict_{\mathrm{HM}}^{\mathrm{FIFO}}$) *After accessing $2k-1$ pairwise different elements in a k-way FIFO set, the set contains only elements from these $2k-1$ accesses. This bound is tight.*

Proof Using Lemma 4 with $i = 2k-1$ gives $e_{\mathrm{HM}}^{\mathrm{FIFO}}(k) \leq 2k-1$. The following example shows the tightness. The access sequence $\langle x_1, \ldots, x_{k-1}, y_1, \ldots, y_{k-1} \rangle$ of

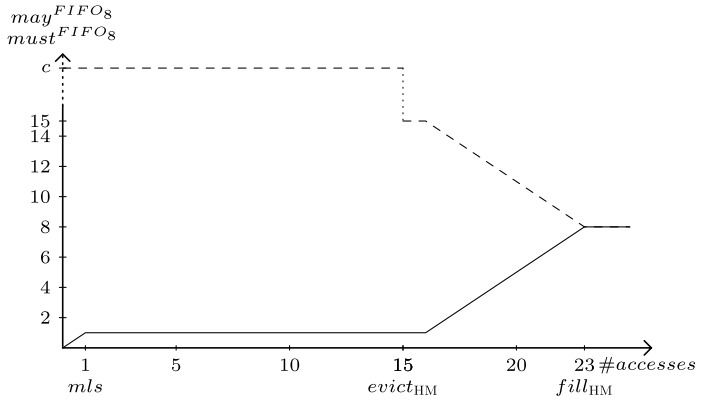


Fig. 4 Evolution of may- and must-information of a 8-way FIFO cache set. c is the number of blocks that can be mapped to the cache set

length $2k-2$ conducted on the initial cache-set state $[z, x_1, \ldots, x_{k-1}]$ results in the state $[y_{k-1}, \ldots, y_1, z]$. Since z survived, $e_{\mathrm{HM}}^{\mathrm{FIFO}}(k) > 2k-2$. □

Theorem 2 ($fill_{\mathrm{HM}}^{\mathrm{FIFO}}$) *One needs at most* $3k-1$ *accesses for any initial cache-set state to reach a completely known cache-set state. This bound is tight.*

Proof Theorem 1 states that after $2k-1$ accesses no more hits can occur. Since the next k accesses will be misses, $3k-1$ is a bound on $f_{\mathrm{HM}}^{\mathrm{FIFO}}(k)$. It is also a tight bound as shown by a similar example as in the proof of Theorem 1. Again, assume initial cache-set 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-set state $[w_{k-1}, \ldots, w_1, y_{k-1}]$, which does not contain z, $f_{\mathrm{HM}}^{\mathrm{FIFO}}(k) > 3k-2$. □

Theorem 3 (mls^{FIFO}) *The minimum life-span of an element in a FIFO-cache is* 1.

Proof Since the queue is not changed on a hit, the element just accessed may reside at the end of the queue. Thus, it may be evicted with the next access. □

As in the LRU-case we have determined the evolution of must- and may-information experimentally. Figure 4 illustrates the results. Disappointingly from a predictability point-of-view, must-information exceeding the minimal life-span of 1 is only attained after 17 accesses.

7 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 remaining 0 bit of a sets status bits is set to 1, all

other bits are reset to 0. This asymmetry in the last bit set to 1 will play a role as we will see later. At cache misses, the line with lowest index (in our representation the left-most) whose MRU-bit is 0 is replaced.

We represent a sample 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 would yield a cache miss and the new state $[e, b, c, d]_{1101}$. Accessing d leaves the state unchanged. A hit on c forces a reset of the MRU-bits: $[e, b, c, d]_{0010}$.

Theorem 4 ($evict_{\mathrm{M}}^{\mathrm{MRU}}$ and $evict_{\mathrm{HM}}^{\mathrm{MRU}}$)

$$evict_{\mathrm{M}}^{MRU}(k) = evict_{\mathrm{HM}}^{MRU}(k) = 2k - 2$$

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 $evict_{\mathrm{HM}}^{\mathrm{MRU}}$ and a lower bound for $evict_{\mathrm{M}}^{\mathrm{MRU}}$. This suffices to prove the tightness for both, since by definition $evict_{\mathrm{M}} \leq evict_{\mathrm{HM}}$.

For the lower bound, consider the initial cache-set 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 the elements $y_1, \ldots, y_{k-2}$ resulting in the state $s'' = [z_1, \ldots, z_{k-2}, y_{k-1}, x_k]_{1\ldots110}$. x_k is still part of the set proving $evict_{\mathrm{M}}^{\mathrm{MRU}}(k) > 2k - 3$.

For the upper bound, notice that at some point during any k pairwise different 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 pairwise different accesses. □

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

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

Proof Consider an access sequence of pairwise different accesses. After at most $2k - 2$ accesses there will be only misses. Therefore a cache-set 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 $fill^{\mathrm{MRU}}(k)$.

Lemma 5 *Consider an MRU cache-set 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 set 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 cache lines will be accessed before the next reset and thus before e is replaced. □

Theorem 6 *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.*

$$\widetilde{fill}_{\mathrm{M}}^{\mathrm{MRU}}(k) := \min\{n \mid must_{\mathrm{M}}^{p_k}(n) = k-1\} = 2k-4.$$

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-set state fits the requirements of Lemma 5 proving the theorem for this case. Otherwise, the reset takes place after at most $k-2$ accesses. $k-2$ additional accesses are sufficient due to Lemma 5 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 $\langle y_1, \ldots, y_{k-2}\rangle \circ \langle z_1, \ldots, z_{k-3}\rangle$: $[x_1, \ldots, x_k]_{0\ldots011} \to [y_1, \ldots, y_{k-2}, x_{k-1}, x_k]_{0\ldots0100} \to [z_1, \ldots, z_{k-3}, y_{k-2}, x_{k-1}, x_k]_{1\ldots100}$. $y_{k-2}, z_1, \ldots, z_{k-3}$ are the last $k-2$ misses but neither x_{k-1} nor x_k which are still in the cache set belong to the last $k-1$ misses. □

Theorem 7 *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.*

$$\widetilde{fill}_{\mathrm{HM}}^{\mathrm{MRU}}(k) := \min\{n \mid must_{\mathrm{HM}}^{p_k}(n) = k-1\} = 3k-4.$$

Proof Due to our general assumption about pairwise different 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 5 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-set 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 tightness for $k > 2$. Note that for $k = 4$ the accesses $v_1, \ldots, v_{k-4}$ and the MRU-bit prefixes $0 \ldots 0$ and $1 \ldots 1$ do not exist. □

Theorem 8 (mls^{MRU}) *The minimum life-span of an element in a MRU-cache is* 2.

Proof The MRU-bit of an accessed element e is always set to 1 resulting in $mls^{\mathrm{MRU}}(k) > 1$. But the next access may reset all the MRU-bits. If e is the left-most element it will be replaced with the next access, which yields $mls^{\mathrm{MRU}}(k) = 2$. □

The evolution of may- and must-information is depicted in Fig. 5. As complete must-information is never attained, the must-curve peaks at 7. Interestingly, may-information never drops from the $14 = 2k - 2$ memory blocks that are reached after *evict* accesses. This can be explained quite easily: the element that causes the reset of the MRU-bits remains in the set for $2k - 2$ further accesses. Due to the unknown initial state any access could have caused the reset. This behavior is in contrast to that of LRU, FIFO, and PLRU, where eventually only the last k accessed elements may reside in a cache set.

8 PLRU caches

PLRU (Pseudo-LRU) is a tree-based approximation of the LRU policy. It arranges the cache lines at the leaves of a tree with $k-1$ "tree bits" pointing to the line to be replaced next. A 0 indicating the left subtree, a 1 indicating the right. See Fig. 6 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. This property 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-set state*. With no invalid lines, equivalent cache-set states with same content and same order of replacements can be obtained by interchanging neighboring subtrees and flipping the corresponding tree bit. We represent a concrete cache set by the equivalent one with all tree bits set to 1. For instance the concrete cache-set state $[a, b, c, d]_{010}$ with tree bits 010 in Fig. 6 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 9 (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 $p_b = pre \circ \overline{p_1} \circ post_b$, where $|p_1|$, 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 $0 \dots 01 \circ post_a$.*

Definition 1 (Miss replacement distance) The *miss replacement distance* $mrd(e)$ of an element e is the minimum number of consecutive misses that are necessary to evict an element from a cache set q. For elements $e \notin q$ we define $mrd(e) = 0$.

Lemma 6 (Miss replacement distance) *A cache line e with access path $p_1 \dots p_n$ has miss replacement distance $mrd(e) = \overline{p_n \dots p_1} + 1$ assuming no invalid lines.*

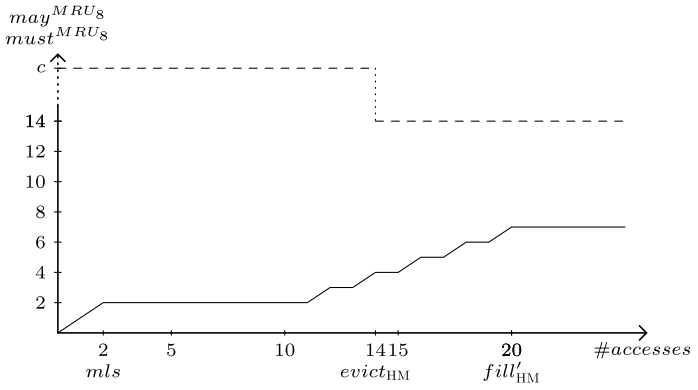
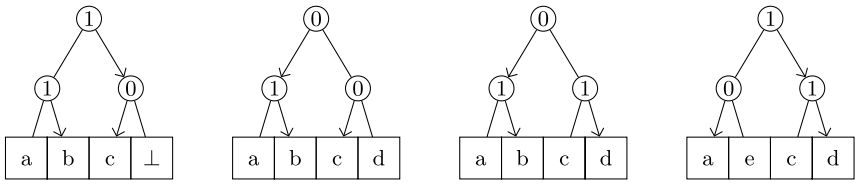


Fig. 5 Evolution of may- and must-information of a 8-way MRU cache set. c is the number of blocks that can be mapped to the cache set. Note that complete must-information can not be obtained, thus $fill'$



(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. 6 Three accesses to a set of a 4-way set-associative PLRU cache: 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 Assuming no invalid lines, all misses will go to access path $1 \ldots 1$. Each miss decrements $\overline{p_n \ldots p_1}$ by 1 for $p_1 \ldots p_n \neq 1 \ldots 1$: consider the dissection of $p_1 \ldots p_n$ into $1 \ldots 1 0 p_{\text{post}}$. A miss updates $p_1 \ldots p_n$ to $0 \ldots 0 1 p_{\text{post}}$ by Observation 9. For $\overline{p_n \ldots p_1}$ this means going from $\overleftarrow{\overline{p_{\text{post}}}} 10 \ldots 0$ to $\overleftarrow{\overline{p_{\text{post}}}} 01 \ldots 1$. □

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

Theorem 10 (mls^{PLRU}) *The minimum life-span of an element in a PLRU-cache is* $mls(k) = \log_2 k + 1$. *In other words, the last* $\log_2 k + 1$ *accesses to a cache set always reside in the set.*

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 9 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. □

8.1 Eviction

Theorem 11 ($evict_{\text{M}}^{\text{PLRU}}$)

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

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 proof is easy. It is a simple consequence of Lemma 6 that 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 6 the line with access path $p_1 \dots p_n$ will be replaced after $\overline{p_n \dots p_1} + 1$ consecutive misses, i.e. the number of trailing 0s in $p_1 \dots p_n$ mainly determines the miss 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

$$\begin{aligned}
\overline{\underbrace{0\dots0}_{\lfloor \log_2 k' \rfloor}\,10\dots0} + 1 &= \underbrace{1\dots1}_{\lfloor \log_2 k' \rfloor}\,01\dots1 + 1 \\
&= \underbrace{1\dots1}_{\lfloor \log_2 k' \rfloor + 1}\,0\dots0 = \underbrace{1\dots1}_{\log_2 k} - \underbrace{1\dots1}_{\log_2 k - (\lfloor \log_2 k' \rfloor + 1)} \\
&= (2^{\log_2 k} - 1) - (2^{\log_2 k - (\lfloor \log_2 k' \rfloor + 1)} - 1) = k - \frac{k}{2^{\lfloor \log_2 k' \rfloor + 1}}.
\end{aligned}$$

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

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

with

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

This proves the given $evict_{\mathrm{M}}^{\mathrm{PLRU}}(k)$ to be an upper bound. To prove its tightness we can give access sequences and initial cache configurations that exactly reach the bounds. Assume $[\bot_1, \dots, \bot_{k-k'}, x_1, \dots, x_{k'}]$ with arbitrary tree bits as the initial configuration. Then, the access sequence $\langle y_1, \dots, y_{k-k'} \rangle$ results in the normalized cache-set state $[y_{i_1}, \dots, y_{i_{k'}}, x_{i_1}, \dots, x_{i_{k'}}, y_{i_{k'+1}}, \dots, y_{i_{k-k'}}]^{\cong}$. Since k' is a power of two the $x_{i_1}, \dots, x_{i_{k'}}$ make up a complete subtree. Therefore, they are not torn apart by accessing other lines in the normalized representation. Furthermore, $y_{k-k'}$ fills $\bot_{k-k'}$ which is adjacent to $x_{i_1}, \dots, x_{i_{k'}}$, moving the x_i-subtree to second position from the left. Observe that the access path $0\dots01\underbrace{0\dots0}_{\log_2 k'}$ leads to x_{i_1}. By Lemma 6, 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.

For the case of hits and misses we need a simple lemma that relates the number of accesses to the two halves of a cache set:

Lemma 7 *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 due to the accesses in the sequence being pairwise different. But every further access to c_2 must then be directly preceded by at least one access to c_1 again yielding $d \leq k$. $(h_1+m_1)-(h_2+m_2) \leq k$ by a similar argument. □

Theorem 12 ($evict^{\mathrm{PLRU}}_{\mathrm{HM}}$) *It takes at most $\frac{k}{2}\log_2 k+1$ pairwise different 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 of accesses needed to evict 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 7 this is bounded by $2z(k)+k-1$.

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

To prove tightness assume a worst-case initial cache-set 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 contents of the cache set 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 sets states and all access sequences of length 2 are worst case initial cache-set states serving as a basis for the recursion. □

8.2 Fill

Theorem 13 ($fill_{\mathrm{M}}^{\mathrm{PLRU}}$) *After at most* $fill_{\mathrm{M}}^{PLRU}(k) = 2k - 1$ *misses the cache-set state is completely known. This bound is tight for* $k > 2$. *For* $k = 2$, 2 *is an obvious tight bound for* $fill_{\mathrm{M}}^{PLRU}$.

Proof At most k misses can go into invalid lines. The last of these accesses resides in the line with access path $0 \ldots 0$ in the normalized cache set. According to Lemma 6, it will be evicted after k further misses, i.e. the $k - 1$ subsequent misses fill up the cache set. Further misses result in a FIFO behavior. The following example proves tightness: assume the initial cache-set state $c = [\bot_1, \ldots, \bot_k]$ 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 path $0 \ldots 0$. 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 9, $x_{\frac{k}{2}}$ has access path $10 \ldots 0$ after $\langle x_{\frac{k}{2}+1}, \ldots, x_k \rangle$. By Lemma 6, it will take $1 \ldots 10 + 1 = k - 1$ further misses to eliminate it, after $k - 2$ accesses it is still in the cache set. Thus, the cache set does not consist of the last k accessed elements, in particular it has not stabilized yet. □

Lemma 8 *If it takes* $evict_{\mathrm{HM}}^{PLRU}(k)$ *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. □

Theorem 14 ($fill_{\mathrm{HM}}^{\mathrm{PLRU}}$) *After at most* $\frac{k}{2} \log_2 k + k - 1$ *pairwise different accesses the PLRU cache-set state is completely known. This bound is tight.*

Proof We want to prove the given bound based on our results for $evict_{\mathrm{HM}}^{\mathrm{PLRU}}(k)$. The difference $fill_{\mathrm{HM}}^{\mathrm{PLRU}}(k) - evict_{\mathrm{HM}}^{\mathrm{PLRU}}(k)$ is $k - 2$. Since the last access to a set always resides in the left-most position with access path $0 \ldots 0$, $k - 1$ additional misses suffice to fill the set due to Lemma 6. This still leaves us one short of the given bound if eviction took exactly $evict_{\mathrm{HM}}^{\mathrm{PLRU}}(k)$ steps. In that case, however, the last two accesses must have gone to different halves due to Lemma 8. Thus, they have access paths $0 \ldots 0$ and $10 \ldots 0$. Due to Lemma 6 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 $e_{\mathrm{HM}}^{\mathrm{PLRU}}(k)$. Let $s = \langle x_1, \ldots, x_e \rangle$ be this worst-case access sequence (assuming the same initial cache-set state). Let $|$ denote the center of the cache set. Then $s' = \langle x_1, \ldots, x_{e-2}, h \rangle \circ \langle y_1, \ldots, y_{k-2} \rangle$ of length $evict_{\mathrm{HM}}^{\mathrm{PLRU}}(k) + k - 3$ results in the intermediate cache-set state $[h, \ldots, x_{e-2}, \ldots, |x_{e-3}, \ldots]^{\cong}$. The final cache-set state is $[y_{i_1}, \ldots, y_{i_{k-2}}, x_{e-3}]^{\cong}$.

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 set.

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 6 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-set state depicted above, which is not yet stabilized. □

The evolution of may- and must-information for a PLRU-set of associativity $k = 8$ is depicted in Fig. 5. As in every policy, must-information initially rises up to $mls(k)$ and reaches k after $fill(k)$ accesses; may-information drops to *evict* after *evict* accesses. The further development of both curves is less uniform than in the other cases, which might be attributed to the more complicated policy.

9 Related work

Sleator and Tarjan (1985) consider replacement policies from a different point of view. They investigate the amortized efficiency of the list update and paging rules LRU, FIFO, LIFO, and LFU. As a reference they take Belady's (1966) optimal offline policy OPT. They show that any online algorithm 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. Their work concerns theoretical performance limits rather than predictability of replacement policies.

Al-Zoubi et al. (2004) perform measurements using the SPEC CPU2000 benchmarks, comparing 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 insights concerning average-case performance in practice. It does however, not deal with predictability.

Heckmann et al. (2003) 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 an 8-way set-associative PLRU cache set which is strongly related to our Theorem 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.

[1] This is due to the construction of our former worst-case example, cf. Theorem 12.

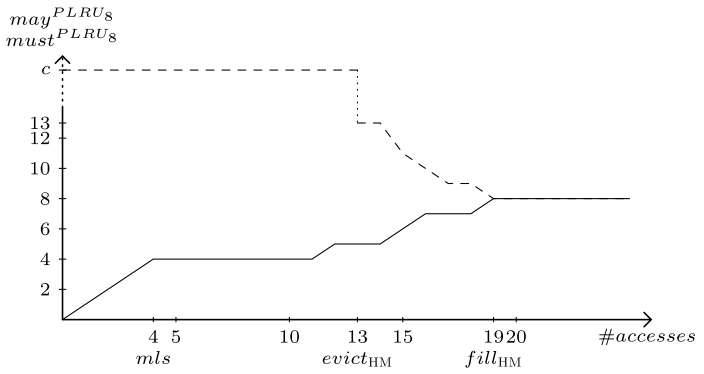


Fig. 7 Evolution of may- and must-information of a 8-way PLRU cache set. c is the number of blocks that can be mapped to the cache set

The manual of the PowerPC 75x series (Freescale Semiconductor Inc. 2002) gives the number of uniquely addressed misses to flush an 8-way PLRU cache set used in these CPUs, which is an instance of $evict_{\mathrm{M}}^{\mathrm{PLRU}}$.

Putting it all together, Al-Zoubi et al. (2004) provide empirical performance results whereas Sleator and Tarjan (1985) present a theoretical performance analysis that is independant of any particular benchmark.

In contrast to performance, predictability in the sense of this paper is concerned with the obtainable precision of provable upper and lower bounds on execution times. Static analysis is used to determine such bounds. Heckmann et al. (2003) provide specific static cache analyses for several replacement policies and compare their precision. Our work presents the theoretical limits of *any* static cache analysis.

10 Conclusions and future work

An important part in the design of hard real-time systems is the proof of timeliness, 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 popular cache replacement policies. To this end, we introduced the metrics *evict* and *fill* and determined their values.

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 $evict_{\mathrm{HM}}(k)$ and $fill_{\mathrm{HM}}(k)$, FIFO and MRU exhibit linear growth in terms of k, while PLRU grows super-linearly. However, instantiating k with the common values 4 and 8 shows a different picture, see Table 2. Here, PLRU even fares slightly better than FIFO and MRU. Yet, compared to 8-way LRU, PLRU, MRU, and FIFO take more than twice as long to regain complete information. In particular, this differs from the worst-case *performance* results obtained in Sleator and Tarjan (1985), where FIFO and LRU fared equally well.

Table 1 Summary of the main results for all policies

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

†See Theorems 6 and 7

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

Policy	$k=4$					$k=8$				
	e_{M}	f_{M}	e_{HM}	f_{HM}	mls	e_{M}	f_{M}	e_{HM}	f_{HM}	mls
LRU	4	4	4	4	4	8	8	8	8	8
FIFO	4	4	7	11	1	8	8	15	23	1
MRU	6	$\infty/4$	6	$\infty/8$	2	14	$\infty/12$	14	$\infty/20$	2
PLRU	5	7	5	7	3	12	15	13	19	4

Our analysis of the evolution of may- and must-information further substantiates the findings: MRU and even more so FIFO should not be considered for use in hard-real time systems. These results support previous practical experience in static cache analysis (Heckmann et al. 2003).

The metrics allow us to investigate the precision of different analyses. Does an analysis ever regain any may- or complete 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?

Future work could drop the restriction that all elements of access sequences are different. This could allow for the construction of precise and efficient (as possible) cache analyses, as we are now aware of the limits. 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? If perfect analyses turns out to be too expensive, our results on the minimal life-span suggest an alternative.

Acknowledgements 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.

We would like to thank Raimund Seidel, Kurt Mehlhorn, Sebastian Hack, and the anonymous referees for helpful comments and fruitful discussions.

References

Al-Zoubi H, Milenkovic A, Milenkovic M (2004) 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, New York, pp 267–272

Belady L (1966) A study of replacement algorithms for a virtual storage computer. IBM Syst J 5:78–101

Ferdinand C, Wilhelm R (1999) Efficient and precise cache behavior prediction for real-time systems. Real-Time Syst 17(2–3):131–181

Freescale Semiconductor Inc (2002) MPC750 RISC Microprocessor User Manual, Section 3.5.1. http://www.freescale.com/files/32bit/doc/ref_manual/MPC750UM.pdf

Heckmann R, Langenbach M, Thesing S, Wilhelm R (2003) The influence of processor architecture on the design and the results of WCET tools. Proc IEEE 91(7):1038–1054

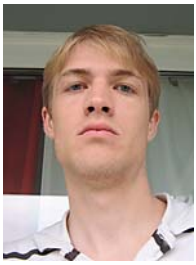
Langenbach M, Thesing S, Heckmann R (2002) Pipeline modeling for timing analysis. In: Proceedings of the static analyses symposium (SAS), vol 2477, Madrid, Spain

Malamy A, Patel R, Hayes N (October 1994) Methods and apparatus for implementing a pseudo-LRU cache memory replacement scheme with a locking feature. United States Patent 5029072

Sleator DD, Tarjan RE (1985) Amortized efficiency of list update and paging rules. Commun ACM 28(2):202–208

Thesing S (2004) Safe and precise WCET determinations by abstract interpretation of pipeline models. PhD thesis, Saarland University

Thiele L, Wilhelm R (2004) Design for timing predictability. Real-Time Syst 28(2–3):157–177



Jan Reineke is a PhD student in Computer Science at Saarland University, where he obtained a Master's degree in 2005. Previously, he was an undergraduate student in Computer Science at the University of Oldenburg until 2003.

Daniel Grund is a PhD student at Saarland University supported by a scholarship of the German Research Foundation. In 2005 he obtained his Diploma degree in Computer Science, with distinction, from the University of Karlsruhe.

Christoph Berg graduated in Computer Science and Physics at Saarland University in 2001. Then he worked as research assistant at the Compiler Design Lab. He is now a systems and database consultant at credativ GmbH.

Reinhard Wilhelm is Professor of Computer Science at the University of the Saarland in Germany. Professor Wilhelm acts as Scientific Director of the International Conference and Research Centre for Computer Science at Schloss Dagstuhl. He is an ACM Fellow. In 2007, he received the Prix Gay-Lussac-Humboldt. He studied mathematics, mathematical logic, physics and computer science at the University of Münster, the Technical University of Munich and Stanford University. He obtained a Diploma degree in Mathematics and a Dr. rer. nat. at TU Munich in 1971 and 1977.