Sensitivity of Cache Replacement Policies

JAN REINEKE, Saarland University, Saarbrücken
DANIEL GRUND, Saarland University, Saarbrücken

The sensitivity of a cache replacement policy expresses to what extent the execution history may influence the number of cache hits and misses during program execution. We present an algorithm to compute the sensitivity of a replacement policy. We have implemented this algorithm in a tool called RELACS that can handle a large class of replacement policies including LRU, FIFO, PLRU, and MRU. Sensitivity properties obtained with RELACS demonstrate that the execution history can have a strong impact on the number of cache hits and misses if FIFO, PLRU, or MRU is used. A simple model of execution time is used to evaluate the impact of cache sensitivity on measured execution times. The model shows that measured execution times may strongly underestimate the worst-case execution time for FIFO, PLRU, and MRU.

Categories and Subject Descriptors: B.3.2 [**Memory Structures**]: Design Styles—*Cache Memories*; B.3.8 [**Memory Structures**]: Performance and Reliability; C.4 [**Computer Systems Organization**]: Performance of Systems—*Measurement techniques*

General Terms: Measurement, Performance, Verification

Additional Key Words and Phrases: Measurement-based timing analysis, cache performance, replacement policy, worst-case execution time, WCET analysis, predictability

ACM Reference Format:
Reineke, J., Grund, D. 2011. Sensitivity of Cache Replacement Policies. ACM Trans. Embedd. Comput. Syst. 9, 4, Article 39 (March 2010), 18 pages.
DOI = 10.1145/0000000.0000000 http://doi.acm.org/10.1145/0000000.0000000

1. INTRODUCTION

Caches are commonly employed to hide the latency gap between memory and the CPU by exploiting locality in memory accesses. On today's microarchitectures a cache miss may take several hundred CPU cycles. Future architectures are expected to exhibit even larger cache miss penalties. Therefore, the cache performance has a strong and increasing influence on a system's overall performance.

In order to fulfill stringent performance requirements, caches are now also used in hard real-time systems. In such systems, guarantees — in the form of lower and upper bounds — have to be given concerning the worst-case execution times (WCET) of tasks. To obtain tight bounds on the execution time of a task, timing analyses *must* take into account the cache architecture. In general, execution times of tasks vary depending on inputs and the execution history of the hardware, i.e., its state. On modern microarchitectures, a large part of this variation can be attributed to the cache performance. At the level of individual instructions, the influence of the hardware state is particularly obvious: cache misses, pipeline stalls, etc. introduce great variance into the execution time of an instruction. It is expected that some of the variances in ex-

Author's addresses: J. Reineke, Department of Electrical Engineering and Computer Sciences, University of California, Berkeley; D. Grund, Department of Computer Science, Saarland University, Saarbrücken.

ecution times of multiple instructions cancel each other out, i.e. worst-cases do not coincide [Bernat et al. 2002]. In addition, one may expect different hardware states to eventually converge. This is true for simple pipelines [Wilhelm et al. 2009]. In many cache architectures and complex pipelines, however, this is not the case [Engblom and Jonsson 2002; Berg 2006].

Different methods have been proposed for timing analysis [Wilhelm et al. 2008]; measurement[1] [Petters 2002; Bernat et al. 2002; Wenzel 2006] and static analysis [Ferdinand et al. 2001; Theiling et al. 2000] being the most prominent. Both methods compute estimates of the worst-case execution times for program fragments like basic blocks. If these estimates are correct, i.e. they are upper bounds on the worst-case execution time of the program fragment, they can be combined to obtain an upper bound on the worst-case execution time of the task. This combination takes into account user-annotated or automatically computed loop bounds.

While using similar methods in the combination of execution times of program fragments, the two methods take fundamentally different approaches to compute these times:

— Static analyses based on abstract models of the underlying hardware compute invariants about the set of all execution states at each program point under *all* possible initial hardware states and inputs and derive upper bounds on the execution time of program fragments based on these invariants.
— Measurement executes each program fragment on a *subset* of the possible initial hardware states and inputs. The maximum of the measured execution times is in general an underestimation of the worst-case execution time.

If the abstract hardware models are correct, static analysis computes safe upper bounds on the WCETs of program fragments and thus also of tasks. However, creating abstract hardware models is an error-prone and laborious process, especially if no precise specification of the hardware is available. Recent work [Schlickling and Pister 2010] explores automation of the creation of abstract hardware models given a concrete VHDL model.

The advantage of measurement over static analysis is that it is more easily portable to new architectures, as it does not rely on an abstract model of the architecture. In addition it may compute more precise estimates of the WCET. On the other hand, soundness of measurement-based approaches is often hard to guarantee. Measurement would trivially be sound if all initial hardware states and inputs would be covered. Due to their huge number this is usually not feasible. Instead, only a subset of the hardware states and inputs can be considered in the measurements, and in some cases it may be possible to identify worst-case initial hardware states.

Relatively simple architectures without any performance-enhancing features like pipelines, caches, etc., exhibit the same timing independently of the initial state. For such architectures, measurement-based timing analysis is sound [Wenzel 2006]. [Deverge and Puaut 2005] and [Wenzel 2006] propose to lock the cache contents [Puaut and Decotigny 2002; Vera et al. 2003] and to flush the pipeline at program points where measurement starts. This is not possible on all architectures and it also has a detrimental effect on both the average- and the worst-case execution times of tasks. In this paper, we study whether safe measurement-based timing analysis is possible in the presence of "unlocked" caches. To this end, we introduce the notion of sensitivity of a cache replacement policy.

[1]Measurement-based timing analysis as discussed here is also referred to as hybrid measurement-based timing analysis as opposed to end-to-end measurement-based analysis.

Sensitivity of a cache replacement policy expresses to what extent the execution history, i.e. the initial hardware state, of the cache may influence the number of cache hits and misses during program execution. We first describe a tool to automatically compute sensitivity properties for a large class of cache replacement policies, including LRU, FIFO, PLRU, and MRU, employing techniques described in [Reineke and Grund 2008]. However, our main contributions besides the introduction of sensitivity are the application of the analysis to relevant policies and the interpretation of the analysis results w.r.t. measurement-based timing analysis: Analysis results demonstrate that the initial state of the cache can have a strong impact on the number of cache hits and misses during program execution if FIFO, PLRU, or MRU replacement is used. A simple model of execution time is used to evaluate the impact of cache sensitivity on measured execution times. The model shows that underestimating the number of misses as strongly as is possible for FIFO, PLRU, and MRU may yield worst-case-execution-time estimates that strongly underestimate the real WCET. In a slightly modified analysis we confirm that the "empty cache is worst-case initial state" assumption is wrong for FIFO, PLRU, and MRU. For these policies, no program- and input-independent worst-case initial cache state exists. On the other hand, our analysis results show that LRU lends itself well to measurement- or simulation-based approaches as the influence of the initial cache state is minimal and the empty cache is indeed the worst-case initial cache state independently of the program to be executed and its inputs.

1.1. Outline

The following section reviews some basics about caches. In Section 3 we formally introduce the notion of sensitivity. In Section 4 we describe how to compute sensitive ratios automatically. Our results are presented in Section 5. Their impact on measured execution times is evaluated in Section 6. Consequences of our results are discussed in Section 7.

2. CACHES

Caches are very fast but small memories that store a subset of the main memory's contents.

To reduce traffic and management overhead, the main memory is logically partitioned into a set of *memory blocks* B 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. The number of such equally sized cache sets s, 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. Caches with associativity $k = 1$ are called *direct-mapped*. Caches consisting of a single cache set, i.e. $s = 1$, are called *fully-associative*.

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. While relatively rare, there are set-associative

$a, b, c \in B$	the set of memory blocks
$\langle b, c, d \rangle, s \in S = B^*$	the set of finite access sequences
$P, Q \in \mathit{Policy}$	the class of replacement policies
$[b, e, c, f]_P, p \in C^P$	the set of reachable cache states of policy P

Fig. 1. Domains and Notations. On the left we list metavariables for elements of the respective domains as well as example elements. For instance, we use s as a metavariable for access sequences and $\langle b, c, d \rangle$ is an example sequence. $[b, e, c, f]_P$ denotes a cache state of policy P whose cache lines contain blocks b, e, c, f.

$\mathit{update}_P(q, s) : C^P \times S \to C^P$	function computing the cache state after accessing a sequence s, starting in q
$m_P(q, s) : C^P \times S \to \mathbb{N}$	function computing the number of misses incurred by policy P conducting s in state q
$h_P(q, s) : C^P \times S \to \mathbb{N}$	function computing the number of hits of policy P conducting s in state q

Fig. 2. Functions that model the behavior of a replacement policy.

caches, where the cache sets are not independent of each other. An example of such policies is PSEUDO ROUND-ROBIN [Heckmann et al. 2003], which maintains a single replacement counter for all cache sets.

Let us briefly explain the four commonly used families of replacement policies under investigation in the course of the paper:

LRU (least-recently-used) replacement conceptually maintains a queue of length k for each cache set, where k is the associativity of the cache. If an element (a memory block) is accessed that is not 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 (LRU) 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. It is used in the INTEL PENTIUM I and the MIPS 24K/34K.

FIFO (first-in-first-out) cache sets can also be seen as queues: 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. FIFO is used in the INTEL XSCALE, ARM9, and ARM11. FIFO is also known as ROUND ROBIN.

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. For an in-detail explanation of PLRU consider [Reineke et al. 2007; Al-Zoubi et al. 2004]. It is used in the POWERPC 75X and the INTEL PENTIUM II-IV.

MRU (most-recently-used), also known as PLRU_M [Al-Zoubi et al. 2004], and, in the context of paging, as CLOCK [Silberschatz et al. 2005]. It maintains one *recently-used bit* per cache line. When a cache line is accessed, the bit is set. Upon a cache miss, the block in the first cache line whose recently-used bit is not set is evicted and the recently-used bit is set. Eventually, the final recently-used bit is set to one. At this point, all other bits are reset to zero. According to [Eklöv et al. 2011], it is used in the INTEL NEHALEM architecture (the codename for an INTEL processor microarchitecture, including processors like the INTEL XEON, CORE I5 and I7).

3. SENSITIVITY

In this section, we formally define the notion of sensitivity. Figure 1 introduces important domains and notations used in the following definitions and throughout the paper. Figure 2 introduces functions that model the behavior of a replacement policy. The most important notions are $m_P(q, s)$ and $h_P(q, s)$, which compute the number of misses and hits, respectively, of policy P starting in state q processing access sequence s.

We would like to investigate the influence of the execution history, i.e. the cache state, on the performance of a cache replacement policy. As set-associative caches can be seen as arrays of fully-associative caches, we consider fully-associative caches only. I.e. we are interested in how sensitive a policy is to the particular state a fully-associative cache is in when beginning to process an access sequence.

Definition 3.1 (*Miss-Sensitivity to State*). A policy P is k-miss-sensitive with additive constant c, if

$$m_P(q, s) \leq k \cdot m_P(q', s) + c$$

for all access sequences $s \in S$ and all cache states $q, q' \in C^P$.

The definition captures the maximal influence of the current state of a replacement policy on the future number of misses. Policy P will incur at most k times the number of misses plus constant c on any access sequence starting in state q instead of a given state q'.

Likewise we define hit-sensitivity.

Definition 3.2 (*Hit-Sensitivity to State*). A policy P is k-hit-sensitive with subtractive constant c, if

$$h_P(q, s) \geq k \cdot h_P(q', s) - c$$

for all access sequences $s \in S$ and all cache states $q, q' \in C^P$.

Policy P will induce at least k times the number of hits minus constant c on any access sequence starting in state q instead of state q'.

As mentioned before, in the following, we are studying policies controlling fully-associative caches. The results translate to set-associative caches, where each of the cache sets is independently controlled, as follows: if policy P is k-miss-sensitive (hit-sensitive) with additive (subtractive) constant c, then a set-associative cache consisting of n cache sets, all controlled by policy P, is k-miss-sensitive (hit-sensitive) with additive (subtractive) constant $c \cdot n$.

We sometimes say that a policy is k-sensitive without specifying an appropriate additive (subtractive) constant. In such cases, we implicitly demand that such a constant exists. The following definition is an example of such a case:

Definition 3.3 (*Sensitive Ratio*). The sensitive miss and hit ratios s_P^m and s_P^h of P are defined as

$$s_P^m = \text{inf}\ \{k \mid P \text{ is } k\text{-miss-sensitive}\}$$

$$\text{and } s_P^h = \text{sup}\ \{k \mid P \text{ is } k\text{-hit-sensitive}\}.$$

Our focus will be on computing these sensitive ratios and appropriate additive (subtractive) constants. Why are we interested in sensitive ratios? Consider a policy that is k-miss-sensitive. It is also l-miss-sensitive for $l > k$. However, the former statement is clearly a better characterization of the policy's sensitivity. In this sense, the sensitive ratio is the *best* characterization of the policy's sensitivity. In particular, there are access sequences, such that the ratio between the number of misses (hits) in one state

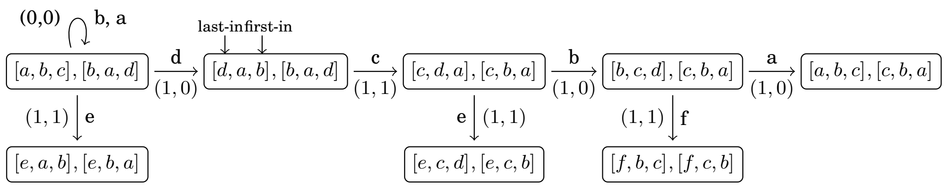


Fig. 3. Running example. Small part of the state space in the computation of sensitivity results for FIFO(3). In the FIFO states, elements are ordered from last-in to first-in. Transitions are labeled with the number of misses incurred. To explain the transitions, we have additionally labeled them with the corresponding accesses. Consider the top-left state $[a, b, c], [b, a, d]$. Accessing a or b incurs hits in both cache states and results in the same state. Accessing d incurs a miss in the first cache state and a hit in the second, resulting in state $[d, a, b], [b, a, d]$.

and the number of misses (hits) in another state approaches the sensitive ratio in the limit. Every policy is by definition 0-hit-sensitive. However, a policy may not be k-miss-sensitive for any k. In that case, we will call it ∞-miss-sensitive. For a policy that is ∞-miss-sensitive, the number of misses starting in one state cannot be bounded by the number of misses starting in another state.

4. COMPUTING SENSITIVE RATIOS

We have developed RELACS[2], a tool that computes sensitive ratios automatically. In the following we will describe the algorithm implemented by RELACS.

To compute sensitive ratios, we construct a transition system, whose states are pairs of cache states, and whose transitions reflect the effect of a memory access on both of the cache states. Paths through this transition system thus correspond to executions of sequences of memory accesses on two cache states, corresponding to the two cache states in the sensitivity definitions. Figure 3 shows a small part of such a transition system.

As the transition system would be infinitely large, assuming an infinite set of memory blocks, we construct a finite quotient structure with respect to an equivalence relation on states that preserves sensitivity properties.

4.1. Induced Transition System

Let us formally define the transition system induced by a policy P.

Definition 4.1 (*Induced Transition System*). A policy P induces a labeled transition system $T_P = (S_P, R_P)$, where

$$S_P = \left\{(q, q') \mid q \in C^P, q' \in C^P\right\},$$

the states, are pairs of cache states of policy P.

$$\begin{aligned} R_P \;=\; \{((p,q),(m_p,m_q),(p',q')) \mid (p,q) \in S_P, a \in B,\\ (p',q') = update_{P,P}((p,q),\langle a\rangle)\\ (m_p,m_q) = m_{P,P}((p,q),\langle a\rangle)\} \end{aligned}$$

is the transition relation, where $update_{P,P}$ and $m_{P,P}$ are lifted to pairs from the $update$ and m functions. Transitions are labeled with the number of misses (0 or 1) incurred by the accesses in the two cache states, respectively.

Sensitive ratios depend on the number of misses (hits) on paths through the transition system:

[2]You can try out RELACS online at http://rw4.cs.uni-saarland.de/~reineke/relacs/.

Definition 4.2 (*Paths*). A path through a labeled transition system $T = (S, R)$, where S is the set of states, $R \subseteq S \times L \times S$ is the set of transitions, and L is a set of labels, is a sequence of labels $\pi = l_1 \dots l_n \in L^n$, such that

$$\exists s_1, \dots, s_{n+1} \in S.\ \forall i \in \{1, \dots, n\}.\ (s_i, l_i, s_{i+1}) \in R.$$

The set of all paths of a transition system T is denoted by $\Pi(T)$.

In our case, labels are pairs (m_p, m_q). The definitions of hit- and miss-sensitivity translate directly to properties of paths of the induced transition system. For instance, a policy P is k-miss-sensitive with additive constant c, if

$$\sum_i \pi(i)_{|1} \leq k \cdot \sum_i \pi(i)_{|2} + c \text{ for all paths } \pi \in \Pi(T_P),$$

where $|1$ and $|2$ select the first and second component of a tuple, respectively.

4.2. Cache States

We assume cache states to be tuples of memory blocks and empty cache lines $B_\perp = B \cup \perp$, possibly equipped with a tuple of status bits $\mathbb{B}^l$:

$$C_k^l \ = \ B_\perp^k \times \mathbb{B}^l$$

So $C^P \subseteq C_k^l$, where k is the associativity of P and l the number of status bits required. For instance, $C^{\mathrm{LRU}(k)} \subseteq C_k^0$, $C^{\mathrm{MRU}(k)} \subseteq C_k^k$, $C^{\mathrm{PLRU}(k)} \subseteq C_k^{k-1}$.

Accesses can have two effects on such states:

— The order of the elements in the tuple is changed, depending *only* on the *position* of the accessed memory block in the tuple and the status bits. In LRU, for instance, the elements are ordered from most to least recently used and no status bits are needed. In FIFO, the elements are ordered from last-in to first-in. Again, no status bits are needed. In contrast, to represent states of MRU, we need k status bits.

— The position in the tuple of the element to be replaced is determined based on the status bits. In our example of LRU, elements are replaced at a fixed position, the right-most, i.e., the least-recently-used. In MRU, the position is determined by the first status bit being 0.

Cache states of all the policies that we consider in this paper, i.e., LRU, PLRU, FIFO, and MRU, behave this way as well as all deterministic replacement policies we can imagine. The main point is that they do *not* base their replacement or update decisions on the particular memory block or its address.

Actual hardware cache implementations do not physically reorder cache lines within cache sets. In hardware implementations, logical ordering information needs to be maintained in status bits. For instance, in case of LRU, at least $\log_2(k!)$ status bits are required to encode any of the $k!$ possible logical orderings of the k cache lines. As a consequence of not being able to efficiently reorder cache lines, the same logical cache state may correspond to several physical cache states. Again, in case of LRU, every logical cache state corresponds to $k!$ equivalent physical cache states. For analysis efficiency it is beneficial not to distinguish cache states that are logically equivalent.

4.3. Quotient Transition System

The induced transition system T_P is infinitely large, if one assumes the set of memory blocks to be infinite. T_P may be prohibitively large even if one assumes a finite number of memory blocks. To overcome this problem we compute a finite quotient structure with respect to an equivalence relation on states that preserves the set of paths of the original transition system. We have previously employed the same equivalence

$$\begin{tikzcd}
\boxed{[a,b,c]_{\text{FIFO}},[b,\perp,e]_{\text{FIFO}}} \arrow[r, phantom, "\approx_h"] \arrow[d, "c"', "(0,1)"] & \boxed{[f,g,l]_{\text{FIFO}},[g,\perp,m]_{\text{FIFO}}} \arrow[d, "h(c)=l"', "(0,1)"] \\
\boxed{[a,b,c]_{\text{FIFO}},[c,b,\perp]_{\text{FIFO}}} \arrow[r, phantom, "\approx_h"] & \boxed{[f,g,l]_{\text{FIFO}},[l,g,\perp]_{\text{FIFO}}}
\end{tikzcd}$$

Fig. 4. In this example, we assume FIFO replacement. $q_1 = [a,b,c],[b,\perp,e] \approx_h q_2 = [f,g,l],[g,\perp,m]$ with an appropriate h-function. An access to c yields a hit in the first element of q_1 and a miss in the second element. The transition is therefore labeled with $(0,1)$. Accessing $h(c) = l$ on q_2 has the same effect in terms of hits and misses. Also, the two resulting states are in the $\approx$ relation by the same function h.

relation to compute the *competitiveness* of one cache replacement policy *relative* to another policy in [Reineke and Grund 2008]. In order to make the paper intelligible by itself, we present a variant of the method presented in [Reineke and Grund 2008] which is specifically tailored to the computation of sensitive ratios. In addition, the algorithm to compute the quotient transition system differs considerably from the one to compute relative competitiveness.

To build a finite quotient transition system, we rely on the following property, which is satisfied by policies representable in the above way and all other cache replacement policies we are aware of: Let $h : B \to B$ be a bijective renaming of the memory blocks and h^* the point-wise extension of h to cache states, that maps $\perp$ (empty cache lines) to $\perp$ and does not modify status bits. Let $q \in C^P$, then

$$update_P(h^*(q), \langle h(a)\rangle) = h^*(update_P(q, \langle a\rangle)) \tag{1}$$

i.e. isomorphic cache states behave the same. Obviously,

$$m_P(h^*(q), \langle h(a)\rangle) = m_P(q, \langle a\rangle), \tag{2}$$

because $h(a)$ is contained in $h^*(q)$ if and only if a is contained in q.

This means that the particular contents of a pair of cache states are irrelevant for the *possible* future ratio of misses. What is important is the relation between the two cache states, i.e. the relative positions of elements contained in both sets. Take the two pairs of cache states $q_1 = [a,b,c],[b,\perp,e]$ and $q_2 = [f,g,l],[g,\perp,m]$. q_1 and q_2 are different regarding the contents of the cache states. Yet, we do not want to distinguish the two states, as they will show the same behavior relative to each other, albeit on different access sequences.

Let h be as above, and $h^\#$ be the point-wise extension of h to pairs of cache states. We identify two states p and q (each is a pair of cache states), denoted $p \approx q$, if they can be transformed into each other by renaming the contents, i.e. if $q = h^\#(p)$ for some h. To indicate a particular feasible renaming function h we also write $p \approx_h q$.

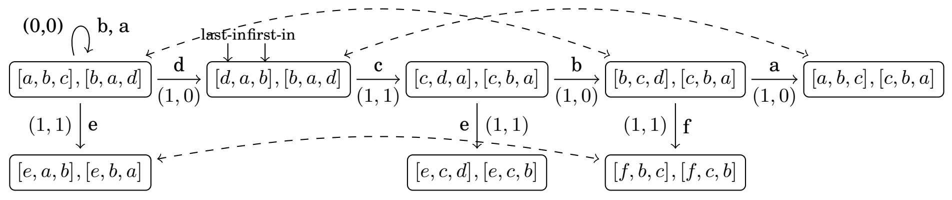
The rationale behind identifying two states $q \approx_h q'$ is that given any access $a \in B$ on q, $h(a)$ will have the "same" effect on q':

$$q \approx_h q' \Rightarrow \begin{cases} m_{P,P}(q, \langle a\rangle) = m_{P,P}(q', \langle h(a)\rangle) \\ update_{P,P}(q, \langle a\rangle) \approx_h update_{P,P}(q', \langle h(a)\rangle) \end{cases} \tag{3}$$

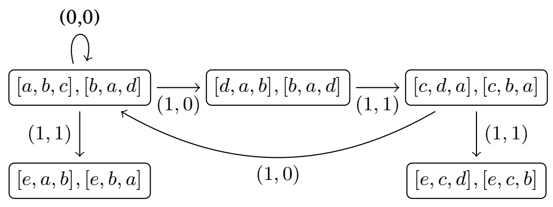
This follows directly from Equation 1 and Equation 2. Figure 4 illustrates Equation 3 with the two example states q_1, q_2 given before.

The relation $\approx$ defines an equivalence on states. It can therefore be used to partition the states of S_P into equivalence classes. This induces a quotient transition system $\overline{T}_P = (\overline{S}_P, \overline{R}_P)$, where $\overline{S}_P \subset S_P$ is a set of unique representatives of the equivalence classes of S_P with respect to $\approx$, and $\overline{R}_P$ is defined as follows:

$$\overline{R}_P = \{(\overline{s}, m, \overline{t}) \mid \exists s,t \in S_P.(s,m,t) \in R_P, s \approx \overline{s}, t \approx \overline{t}\} \tag{4}$$



(a) Dashed lines connect equivalent states according to the equivalence relation $\approx$.



(b) Quotient structure.

Fig. 5. Running example revisited. "Merging" equivalent states in (a) yields the depicted quotient structure in (b).

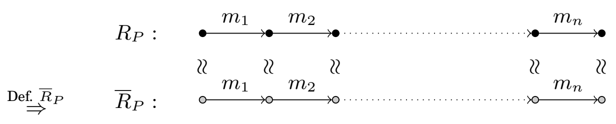
There is a transition $(\overline{s}, m, \overline{t})$ between two representatives iff there is a transition (s, m, t) between two states s and t that are represented by $\overline{s}$ and $\overline{t}$, respectively. Figure 5 illustrates the $\approx$ relation in the running example and shows the resulting quotient structure.

By the following theorem we can safely work with the quotient structure $\overline{T}_P$ instead of T_P when computing sensitivity values.

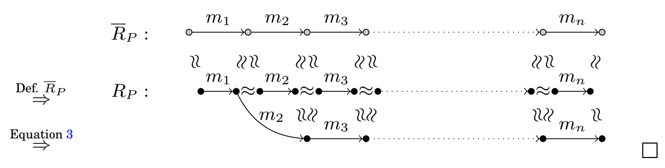
THEOREM 4.3 (PATH EQUIVALENCE). *The transition systems T_P and $\overline{T}_P$ are path equivalent, i.e.,* $\Pi(T_P) = \Pi(\overline{T}_P)$.

PROOF. We need to show $\pi = m_1 \dots m_n \in \Pi(T_P) \Leftrightarrow \pi \in \Pi(\overline{T}_P)$.

"$\Rightarrow$" follows immediately from the definition of $\overline{R}_P$ in Equation 4. Let $s_1, \dots, s_{n+1}$ be such that $(s_i, m_i, s_{i+1}) \in R_P, \forall i \in \{1, \dots, n\}$. Let $\overline{s_1}, \dots, \overline{s_{n+1}}$ be the representants of $s_1, \dots, s_{n+1}$ in $\overline{S}_P$. Then $(\overline{s_i}, m_i, \overline{s_{i+1}}) \in \overline{R}_P, \forall i \in \{1, \dots, n\}$:



"$\Leftarrow$" is just a little more complicated. For each of the edges $(\overline{s_i}, m_i, \overline{s_{i+1}})$ in $\overline{R}_P$ there is a corresponding edge in R_P by definition. By Equation 3 we can construct a path through R_P from these edges as illustrated below:



$\square$

Intuitively, the definition of $\overline{R}_P$ guarantees that $\Pi(T_P) \subseteq \Pi(\overline{T}_P)$. Together, the definition of $\overline{R}_P$ and Equation 3 yield $\Pi(\overline{T}_P) \subseteq \Pi(T_P)$.

OBSERVATION 4.4. *The set of equivalence classes of $\approx$ is finite, as it only depends on the relative positions of elements contained in both sets. In particular, the index of $\approx$ is independent of the number of memory blocks.*

Therefore, the quotient transition system $\overline{T}_P$ is finite. In [Reineke 2008] lower and upper bounds on the index of $\approx$ are derived: it is exponential in the associativity and the number of status bits of the replacement policy. Note that we directly construct $\overline{T}_P$, in particular we never construct the underlying infinite transition system T_P.

4.4. Constructing the Quotient Transition System

Algorithm 1 directly constructs $\overline{T}_P$. It consists of two steps: The computation of $\overline{S}_P$ and the computation of $\overline{R}_P$. In order to construct the quotient transition system on-the-fly, we have to compute unique representatives of the states that we encounter in the construction of the transition system. NORMALIZE(p, q) computes a unique representative in the equivalence relation for pairs of states. We explain how to compute such a representative in Section 4.4.1.

To compute $\overline{S}_P$, the algorithm proceeds by taking a yet *unprocessed* state from the $Unprocessed$ queue and by computing all its successor states until all states have been processed. It starts with a pair of initial states of the policy (i^P, i^P). Instead of computing successors under the same accesses only, as necessary when computing competitive ratios, we have to take into account arbitrary uncorrelated accesses to both cache states. This is because in the case of relative competitiveness only so-called *compatible* pairs of cache states are considered[3], whereas in the case of sensitivity $\overline{S}_P$ consists of all pairs of states of policy P.

$CC_P(q)$ are the contents of state q, e.g. $CC_{\mathbf{FIFO}(4)}([d, a, b, c]) = \{a, b, c, d\}$. $\overline{S}$ denotes the complement of the set S, i.e. $\overline{S} = B \setminus S$, where B is the set of memory blocks.

Note that for a fixed a, NORMALIZE$(update_P(p, \langle a\rangle), update_P(q, \langle b\rangle))$ yields $\approx$-equivalent states for all $b \in \overline{CC_P(p) \cup CC_P(q) \cup \{a\}}$. Similarly, $\approx$-equivalent states arise for a fixed b in NORMALIZE$(update_P(p, \langle a\rangle), update_P(q, \langle b\rangle))$, for all $a \in \overline{CC_P(p) \cup CC_P(q) \cup \{b\}}$. Therefore, it suffices to consider the following five cases of access pairs (a, b), which are implicitly covered by the algorithm:

(1) $(a, b) \in (CC_P(p) \cup CC_P(q)) \times (CC_P(p) \cup CC_P(q))$.
(2) $(a, b) \in (CC_P(p) \cup CC_P(q)) \times \{m_1\}$.
(3) $(a, b) \in \{m_1\} \times (CC_P(p) \cup CC_P(q))$.
(4) $(a, b) = (m_1, m_1)$.
(5) $(a, b) = (m_1, m_2)$.

where $m_1 \in \overline{CC_P(p) \cup CC_P(q)}$ and $m_2 \in \overline{CC_P(p) \cup CC_P(q) \cup \{m_1\}}$.

Once $\overline{S}_P$ has been computed, $\overline{R}_P$ can be computed as follows. The key insight is that NORMALIZE$(update_P(p, \langle a\rangle), update_P(q, \langle a\rangle))$ is equal for all $a \notin CC_P(p) \cup CC_P(q)$. All of these accesses will be misses in both p and q and thus result in $\approx$-equivalent successor states. Therefore, it is sufficient to compute successors under the finite number of accesses $CC_P(p) \cup CC_P(q) \cup \{\text{SELECTONE}(\overline{CC_P(p) \cup CC_P(q)})\}$, where SELECTONE$(S)$ selects an arbitrary element of S.

4.4.1. Computing Unique Representatives. In order to construct the quotient transition system on-the-fly, we have to compute unique representatives of the states that we encounter in the construction of the transition system. Given a well-founded total order $\leq_{B_\bot} \subseteq B_\bot \times B_\bot$ on memory blocks B and empty lines, such that $\bot$ is the least element

[3]See [Reineke 2008] for details.

Algorithm 1: Constructing the Quotient Transition System

```
Input: Policy $P$
Output: Quotient Transition System $\overline{T}_P = (\overline{S}_P, \overline{R}_P)$
begin
    $\overline{S}_P \leftarrow \{\text{NORMALIZE}(i^P, i^P)\}$;
    $Unprocessed \leftarrow [\text{NORMALIZE}(i^P, i^P)]$;
    while $\neg\text{EMPTY}(Unprocessed)$ do
        $(p, q) \leftarrow \text{POP}(Unprocessed)$;
        $m_1 \leftarrow \text{SELECTONE}(\overline{CC_P(p) \cup CC_P(q)})$;
        $m_2 \leftarrow \text{SELECTONE}(\overline{CC_P(p) \cup CC_P(q) \cup \{m_1\}})$;
        foreach $a \in CC_P(p) \cup CC_P(q) \cup \{m_1\}$ do
            $p' \leftarrow update_P(p, \langle a \rangle)$;
            foreach $b \in CC_P(p) \cup CC_P(q) \cup \{m_1, m_2\}$ do
                $(p', q') \leftarrow \text{NORMALIZE}(p', update_P(q, \langle b \rangle))$;
                if $(p', q') \notin \overline{S}_P$ then
                    $\text{PUSH}(Unprocessed, (p', q'))$;
                    $\overline{S}_P \leftarrow \overline{S}_P \cup \{(p', q')\}$;
    $\overline{R}_P \leftarrow \emptyset$;
    foreach $(p, q) \in \overline{S}_P$ do
        foreach $a \in CC_P(p) \cup CC_P(q) \cup \{\text{SELECTONE}(\overline{CC_P(p) \cup CC_P(q)})\}$ do
            $(p', q') \leftarrow \text{NORMALIZE}(update_P(p, \langle a \rangle), update_P(q, \langle a \rangle))$;
            $(m_p, m_q) \leftarrow (m_P(p, \langle a \rangle), m_P(q, \langle a \rangle))$;
            $\overline{R}_P \leftarrow \overline{R}_P \cup \{((p, q), (m_p, m_q), (p', q'))\}$;
```

of $\leq_{B_\perp}$, we define a well-founded partial order $\leq_{S_P} \subseteq S_P \times S_P$ on the states in S_P:

$$(p, q) \leq_{S_P} (p', q') :\Leftrightarrow p <_{C_k^l} p' \vee (p = p' \wedge q \leq_{C_k^l} q'),$$

where cache states in $C^P \subseteq C_k^l$ are as well ordered lexicographically based on $\leq_{B_\perp}$:

$$[m_1, \ldots, m_k]_{b_1 \ldots b_l} \leq_{C_k^l} [m'_1, \ldots, m'_k]_{b'_1 \ldots b'_l} :\Leftrightarrow \exists i > 0 : \forall j < i : m_j = m'_j \wedge m_i \leq_{B_\perp} m'_i \\ \wedge \forall i : b_i = b'_i.$$

Note that $\leq_{S_P}$ is only partial in general as the status bits have to agree for a pair of states to be in $\leq_{S_P}$. However, it is total on $\approx$-equivalent states. Therefore, there is always a *least* state in an equivalence class.

We choose the least state of an equivalence class in this order as the representative of its equivalence class:

$$\overline{S}_P := \{s_P \in S_P \mid \forall s'_P : s_P \approx s'_P \Rightarrow s_P \leq_{S_P} s'_P\}$$

These unique representatives can be efficiently computed by constructing a renaming function h traversing a state from left to right and choosing the minimal memory block still available for the renaming:

Example 4.5. Consider the total order $\leq_M$ to order memory blocks alphabetically: $\perp \leq_{B_\perp} a \leq_{B_\perp} b \leq_{B_\perp} c \leq_{B_\perp} \ldots$. Then, the representative of $[g, f, \perp, h], [g, h, l, \perp]$ can be computed by consecutively renaming the elements of the state:

$$\begin{aligned} & [g, f, \perp, h], [g, h, l, \perp] \rightsquigarrow [a, f, \perp, h], [a, h, l, \perp] \\ \rightsquigarrow & [a, b, \perp, h], [a, h, l, \perp] \rightsquigarrow [a, b, \perp, c], [a, c, l, \perp] \\ \rightsquigarrow & [a, b, \perp, c], [a, c, d, \perp] \end{aligned}$$

4.5. Computation of Sensitive Ratios

Once we have built the quotient transition system, determining the minimal k such that P is miss- (hit-) sensitive amounts to computing the *maximum (minimal) cycle ratio* [Lawler 1966; Ahuja et al. 1993]:

In the setting of hit-sensitivity, we wish to find a cycle through the quotient transition system that minimizes the ratio of hits ("cost") in one component relative to the number of hits ("time") in the other component. In the case of miss-sensitivity, we are looking for a cycle that maximizes the ratio of misses ("cost") in one component relative to the number of misses ("time") in the other component.

The *minimum cycle ratio* problem, also known as the *minimum cost-to-time ratio cycle problem*, is the following: Given a directed graph G with both a cost and a travel time associated with each edge, we wish to find a cycle in the graph with the smallest ratio of its cost to its travel time.

Definition 4.6 (*Minimum Cycle Ratio*). The minimum cycle ratio λ^* of G is

$$\lambda^* = \min_{\text{Cycle } C \in G} \frac{\sum_{\text{Edge } (i,j) \in C} c_{ij}}{\sum_{\text{Edge } (i,j) \in C} \tau_{ij}}$$

where c_{ij} and τ_{ij} are the cost and travel time associated with edge (i, j).

Lawler and Ahuja et al. [Lawler 1966; Ahuja et al. 1993] describe how to solve the minimum-cycle-ratio problem by repeated applications of a negative cycle detection algorithm in polynomial time.

The maximum cycle ratio is defined analogously to the minimum cycle ratio, simply replacing $\min$ by $\max$. Maximum-cycle-ratio problems can be transformed into minimum-cycle-ratio problems by, e.g., changing the sign of the costs associated with all edges. The negation of the minimum cycle ratio of the transformed problem then yields the maximum cycle ratio.

As noted above, we need to compute the maximum cycle ratio of $\overline{T}_P$ to obtain the miss sensitive ratio:

THEOREM 4.7 (MAXIMUM CYCLE RATIO). *The maximum cycle ratio k of $\overline{T}_P$ is equal to the sensitive miss ratio of P.*

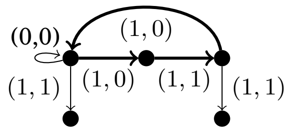
PROOF. We need to show that

(1) P is k-miss-sensitive with some additive constant c.
(2) P is not k'-miss-sensitive with any additive constant c' for $k' < k$.

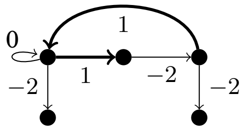
For 1. we need to show

$$\sum_i \pi(i)_{|1} \leq k \cdot \sum_i \pi(i)_{|2} + c \text{ for all paths } \pi \in \Pi(T_P).$$

Any path π can be split into three (possibly empty) parts $\pi = \pi_0\pi_1\pi_2$, such that π_0 and π_2 correspond to acyclic traversals of the state space $\overline{S}_P$ and π_1 corresponds to a cycle in $\overline{S}_P$. Since $\overline{S}_P$ is finite, $|\pi_0| < |\overline{S}_P|$ and $|\pi_2| < |\overline{S}_P|$. For the cycle part π_1 we



(a) Part of the quotient structure $\overline{T}_{\text{FIFO}(3)}$. Maximum cycle ratio $\frac{1+1+1}{0+1+0} = 3$.



(b) Longest path, w.r.t. edge weights $w = m_1 - 3m_2$. All cycles have length less than or equal to 0.

Fig. 6. Running example. Miss-Sensitivity of FIFO(3). FIFO(3) is 3-miss-sensitive with additive constant 3. The small part of $\overline{T}_{\text{FIFO}(3)}$ does not contain an acyclic path of length 3. The longest path, in the example, is only of length 2, whereas the full transition system contains paths of length 3.

know that $\sum_i \pi_1(i)_{|1} \leq k \cdot \sum_i \pi_1(i)_{|2}$. The acyclic paths π_0 and π_2 can be "covered" by an appropriate constant $c \leq 2 \cdot |\overline{S}_P|$.

For 2.: Choose a "cyclic" path π that corresponds to the maximal cycle ratio k. As $k > k'$, $\sum_i \pi(i)_{|1} = k \cdot \sum_i \pi(i)_{|2} > k' \cdot \sum_i \pi(i)_{|2}$. By repeating π appropriately often $\sum_i \pi^n(i)_{|1} > k' \cdot \sum_i \pi^n(i)_{|2} + c'$ for any additive constant c'. □

The proof shows that the additive constant c can only stem from finite acyclic pre- or suffixes of paths. Once the minimum cycle ratio k has been determined the appropriate additive (subtractive) constant c can be determined as follows: in $\overline{T}_P$ one assigns edge weights $w := c_{ij} - k\tau_{ij}$. Then, paths with negative (positive) weight correspond to situations where one component does "worse" than suggested by k for a limited number of steps. Computing the shortest (longest) path through that graph yields the constant c. As k is the minimum (maximum) cycle ratio, that graph has no negative (positive) cycles. As an example, assume P to be 2-miss-sensitive. Then, there will be paths of length one or more, such that one component makes misses and the other does not.

In Figure 6, we illustrate the two steps of our algorithm for our running example and the case of miss-sensitivity. To improve readability of the example, we omit the node labels.

The hit sensitivity of a policy is the minimum cycle ratio of a quotient transition system whose edge weights correspond to the number of hits of the policy, rather than the number of misses, as in the case of miss sensitivity. The quotient transition system for miss sensitivity can be converted into such a transition system by exchanging 0s and 1s in the edge weights. To avoid repetition, we omit the formal definition of this quotient transition system as well as the theorem relating the minimum cycle ratio of this system to the hit-sensitivity of the policy.

5. RESULTS

Using our tool, we have obtained sensitivity results for LRU, FIFO, PLRU, and MRU at associativities ranging from 2 to 8. Note that we have computed the precise sensitive ratios not just upper bounds. I.e. there are arbitrarily long access sequences and pairs of initial states that exhibit the computed hit and miss ratios.

Figure 7(a) depicts our results for the miss-sensitivity of LRU, FIFO, and PLRU. LRU is very insensitive to its state. The difference in misses is bounded by the associativity k. This is unavoidable for any policy, as the initial states may have completely disjoint contents. FIFO, MRU, and PLRU are much more sensitive to their state than LRU.

Depending on its state, FIFO(k) may incur up to k times as many misses. PLRU(2) coincides with LRU(2). For greater associativities, the number of misses incurred

	2	3	4	5	6	7	8
LRU	$1, 2$	$1, 3$	$1, 4$	$1, 5$	$1, 6$	$1, 7$	$1, 8$
FIFO	$2, 2$	$3, 3$	$4, 4$	$5, 5$	$6, 6$	$7, 7$	$8, 8$
PLRU	$1, 2$	–	∞	–	–	–	∞
MRU	$1, 2$	$3, 4$	$5, 6$	$7, 8$	MEM	MEM	MEM

(a) Miss-Sensitive ratio k, and constant c, for FIFO, PLRU, LRU, and MRU.

	2	3	4	5	6	7	8
LRU	$1, 2$	$1, 3$	$1, 4$	$1, 5$	$1, 6$	$1, 7$	$1, 8$
FIFO	$0, 0$	$0, 0$	$0, 0$	$0, 0$	$0, 0$	$0, 0$	$0, 0$
PLRU	$1, 2$	–	$\frac{1}{3}, \frac{5}{3}$	–	–	–	$\frac{1}{11}, \frac{19}{11}$
MRU	$1, 2$	$0, 0$	$0, 0$	$0, 0$	MEM	MEM	MEM

(b) Hit-Sensitive ratio k, and subtractive constant c, for FIFO, PLRU, LRU, and MRU.

Fig. 7. Miss- and Hit-Sensitivity Results. As an example of how this should be read, PLRU(4) is $\frac{1}{3}$-hit-sensitive with subtractive constant $\frac{5}{3}$. ∞ indicates that a policy is not k-miss-competitive for any k. A "–" indicates that the policy is not defined for the respective associativity. In particular, PLRU is only defined for powers of two. MEM indicates that the algorithm ran out of memory on a 2 GB machine.

starting in one state cannot be bounded by the number of misses incurred starting in another state. Of course, the number of misses is always bounded by the length of the access sequence. However, given *only* the number of misses and not the length of the sequence no bound can be given.

As the number of misses may only differ by a constant for LRU, the number of hits may only differ by the same constant. For FIFO, the situation is different: given the number of hits in one state, it is impossible to derive a lower bound on the number of hits in another state. The results for PLRU are only slightly more encouraging than in the miss-sensitivity case. At associativity 8, a sequence may only cause $1/11$ of the number of hits depending on the starting state. See Figure 7(b) for the analysis results.

Summarizing, FIFO, MRU, and PLRU may in the worst case be heavily influenced by the starting state. LRU is very robust: the number of hits and misses is affected in the least possible way.

5.1. Is the Empty Cache the Worst-Case Initial State?

One might suppose that it is still safe to assume an empty cache or equivalently a cache filled with irrelevant data only as the starting state [Petters 2002, page 39ff], assuming that an empty cache were worse than any non-empty cache. This is *not* true for FIFO, MRU, and PLRU. We have performed a second analysis that fixed the reference starting state (q' in the definitions) to be empty. The analysis revealed the same sensitive ratios as in the general case with all additive (subtractive) constants being zero. For LRU, this is in fact a positive result, as it confirms that the empty cache is indeed the worst-case for any access sequence.

This is the example produced by the tool for the miss-sensitivity of FIFO(4):

Example 5.1 (*Miss-sensitivity of* FIFO(4)).
Consider the pair of cache-set states $[b, c, d, e]_{\mathrm{FIFO}}, [\perp, \perp, \perp, \perp]_{\mathrm{FIFO}}$.
The sequence $\langle d, e, a, b\rangle$ leads it to the pair $[a, b, c, d]_{\mathrm{FIFO}}, [b, a, e, d]_{\mathrm{FIFO}}$:

$$[b, c, d, e], [\perp, \perp, \perp, \perp] \xrightarrow{d} [b, c, d, e], [d, \perp, \perp, \perp] \xrightarrow{e} [b, c, d, e], [e, d, \perp, \perp]$$
$$\xrightarrow{a} [a, b, c, d], [a, e, d, \perp] \xrightarrow{b} [a, b, c, d], [b, a, e, d]$$

Now, consider the access sequence $\langle e, d, f, b \rangle$, which leads the pair into the $\approx$-equivalent pair $[b, f, d, e]_{\text{FIFO}}, [f, b, a, e]_{\text{FIFO}}$:

$$\begin{aligned}[a, b, c, d], [b, a, e, d] \xrightarrow{e} [e, a, b, c], [b, a, e, d] \xrightarrow{d} [d, e, a, b], [b, a, e, d]\\ \xrightarrow{f} [f, d, e, a], [f, b, a, e] \xrightarrow{b} [b, f, d, e], [f, b, a, e]\end{aligned}$$

The state that originated from the empty state $[\perp, \perp, \perp, \perp]_{\text{FIFO}}$ incurs only one miss on this sequence, while the other state misses in each of the four accesses. As the resulting states are $\approx$-equivalent there is another sequence that will show the same behavior in the two states and so on.

It has been observed earlier [Berg 2006], that the empty cache is not necessarily the worst-case starting state for PLRU. Our work demonstrates to what extent it may be better than the real worst-case initial state in the case of FIFO, MRU, and PLRU. It turns out that except for the additive (subtractive) constant, starting with an empty cache may be as bad as starting in any other state.

5.2. Pathological Cases?

Of course, it is not very likely to start measurements in a state that minimizes the number of misses for the following access sequence. Yet, it is difficult to associate a particular probability with this event. One should also realize that many states in between the worst and the best case (i.e. even if one does not start in the state that minimizes the number of misses) may still perform significantly better than the worst-case initial state.

6. IMPACT OF RESULTS ON TIMING ANALYSIS

We will try to illustrate on a simplified scenario the impact of the sensitivity results on measured execution times.

To this end, we adopt a simple model of execution time in terms of cache performance of [Hennessy and Patterson 1996]. In this model, the execution time is the product of the clock cycle time and the sum of the CPU cycles (the pure processing time) and the memory stall cycles:

$$\text{Exec. time} = (\text{CPU cycles} + \text{Mem. stall cycles}) \times \text{Clock cycle}$$

The equation makes the simplifying assumption that the CPU is stalled during a cache miss. Furthermore, it assumes that the CPU clock cycles include the time to handle cache hits.

Let CPI_{hit} be the average number of cycles per instruction if no cache misses occur. Then, the CPU cycles are simply a product of the number of instructions IC and CPI_{hit}:

$$\text{CPU cycles} = \text{IC} \times \text{CPI}_{hit}$$

The number of memory stall cycles depends on the number of instructions IC, the number of misses per instruction and the cost per miss, the miss penalty:

$$\begin{aligned}\text{Mem. stall cycles} &= \text{Number of misses} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Misses}}{\text{Instruction}} \times \text{Miss penalty}\\ &= \text{IC} \times \frac{\text{Mem. accesses}}{\text{Instruction}} \times \text{Miss rate} \times \text{Miss Pen.}\end{aligned}$$

Now assume we have measured an execution time of T_{meas} in a system with a 4-way set-associative FIFO cache. By which factor may the "real" worst-case execution time T_{wc} differ from T_{meas} due to different initial states of the cache?

As in the example of Hennessy and Patterson [Hennessy and Patterson 1996, page 386f], modeling a machine similar to the Alpha AXP 21064[4], let the number of memory accesses per instruction be 1.33[5] and let the miss penalty be 50. Further assume that instructions take 2 cycles on average due to pipeline stalls, i.e. the CPI_{hit} is 2. Further assume, the miss rate Miss rate_{meas} during the measurement was 2%. The sensitive miss-ratio of FIFO(4) is 4. Neglecting the additive constant, the worst-case miss rate Miss rate_{wc} could thus be as high as 8%. Plugging the above assumptions into the equations and simplification yields

$$\begin{aligned}\frac{T_{wc}}{T_{meas}} &= \frac{\text{CPI}_{hit} + \frac{\text{Mem. accesses}}{\text{Instruction}} \times \text{Miss rate}_{wc} \times \text{Miss Pen.}}{\text{CPI}_{hit} + \frac{\text{Mem. accesses}}{\text{Instruction}} \times \text{Miss rate}_{meas} \times \text{Miss Pen.}} \\ &= \frac{2 + 1.33 \times 0.08 \times 50}{2 + 1.33 \times 0.02 \times 50} = \frac{7.32}{3.33} \approx 2.2.\end{aligned}$$

So, in the example of a 4-way set-associative FIFO cache, the worst-case execution time may be a factor of 2.2 higher than the measured time only due to the influence of the initial cache state. If PLRU were used as a replacement policy the difference could be even greater. As measurement usually does not allow to determine the miss rate (or simply the number of misses) it is not even possible to add a conservative overhead to the measured execution times to account for the sensitivity to the initial state.

The above analysis considers the impact of cache sensitivity on an individual measurement. Measurement-based timing analysis as described in the literature [Petters 2002; Bernat et al. 2002; Wenzel 2006; Deverge and Puaut 2005] does not advocate end-to-end measurements. Instead, measurements of program fragments are performed and later combined to obtain an estimate of the worst-case execution time of the whole program. The above arguments apply to any of the measurements of program fragments. If the measurement of an important fragment like the body of an inner loop is far off, the estimate for the whole program will as a consequence be far off as well.

7. SUMMARY, CONCLUSION, AND FUTURE WORK

We have introduced a notion of sensitivity of cache replacement policies that captures the influence of the execution history of the cache on the future cache performance. Employing techniques first described in [Reineke and Grund 2008] allows to compute sensitive ratios of a large class of replacement policies, including the four well-known and widely-used families of replacement policies, LRU, FIFO, PLRU, and MRU.

The analysis results revealed great differences among LRU, FIFO, MRU, and PLRU, that yield another argument in favor of using LRU in the design of predictable real-time systems. In the case of FIFO, PLRU, and MRU, the initial state can have a great influence on the number of hits and misses during program execution. Notably, the assumption that an empty cache is the worst-case initial state is wrong for those policies. A simple model of execution time demonstrates the impact of cache sensitivity on measured execution times. If for a given program, the worst-case number of misses exceeds the measured number of misses as strongly as is possible, then measurement-based WCET estimates strongly underestimate the WCET for FIFO, PLRU, and MRU. To obtain safe results by measurement with respect to cache performance the cache contents should be locked as proposed in [Deverge and Puaut 2005; Wenzel 2006; Vera et al. 2003; Puaut and Decotigny 2002], which may have an adverse effect on average- and worst-case execution time.

[4]Modern processor exhibit higher miss penalties than the assumed Alpha AXP. In this regard the following analysis is thus conservative.
[5]Each instruction causes one instruction fetch and possibly data fetches.

Our results hold for arbitrary access sequences. Therefore, they hold for any program. Many programs do not exhibit the worst-case cache behavior. By restricting the possible access sequences it would be possible to obtain smaller sensitive ratios for particular programs. However, computing precise sensitive ratios in such restricted scenarios is more difficult than in the present case as we cannot compute a quotient transition system in a similar way. States that are equivalent in the current setting may not be equivalent if memory accesses are restricted in some way.

Acknowledgements

This work has profited from discussions within the ARTIST2 and ARTIST DESIGN Networks of Excellence. It has been supported by the German Research Foundation (DFG) as part of SFB/TR 14 AVACS, the German-Israeli Foundation for Scientific Research and Development (GIF), and by the European Community's Seventh Framework Programme FP7 under grant agreement number 216008 (Predator). We thank Reinhard Wilhelm for numerous discussions on this work.

REFERENCES

AHUJA, R. K., MAGNANTI, T. L., AND ORLIN, J. B. 1993. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA.

AL-ZOUBI, H., MILENKOVIC, A., AND 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*. ACM Press, New York, NY, USA, 267–272.

BERG, C. 2006. PLRU cache domino effects. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, F. Mueller, Ed. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany.

BERNAT, G., COLIN, A., AND PETTERS, S. M. 2002. WCET analysis of probabilistic hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*. IEEE Computer Society, Washington, DC, USA, 279.

DEVERGE, J.-F. AND PUAUT, I. 2005. Safe measurement-based WCET estimation. In *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, R. Wilhelm, Ed. Internationales Begegnungs- und Forschungszentrum für Informatik (IBFI), Schloss Dagstuhl, Germany, Dagstuhl, Germany.

EKLÖV, D., NIKOLERIS, N., BLACK-SCHAFFER, D., AND HAGERSTEN, E. 2011. Cache pirating: Measuring the curse of the shared cache. In *ICPP*. 165–175.

ENGBLOM, J. AND JONSSON, B. 2002. Processor pipelines and their properties for static WCET analysis. In *Proceedings of the Second International Conference on Embedded Software*. EMSOFT '02. Springer-Verlag, London, UK, 334–348.

FERDINAND, C., HECKMANN, R., LANGENBACH, M., MARTIN, F., SCHMIDT, M., THEILING, H., THESING, S., AND WILHELM, R. 2001. Reliable and precise WCET determination for a real-life processor. In *Embedded Software Workshop*. Vol. 2211. Lake Tahoe, USA, 469 – 485.

HECKMANN, R., LANGENBACH, M., THESING, S., AND WILHELM, R. 2003. The influence of processor architecture on the design and the results of WCET tools. *Proceedings of the IEEE 91,* 7, 1038–1054.

HENNESSY, J. L. AND PATTERSON, D. A. 1996. *Computer Architecture: A Quantitative Approach, 2nd Edition*. Morgan Kaufmann.

LAWLER, E. 1966. Optimal cycles in doubly weighted linear graphs. In *Int'l Symp. Theory of Graphs*. 209–213.

PETTERS, S. M. 2002. Worst case execution time estimation for advanced processor architectures. Ph.D. thesis, Technische Universität München, Munich, Germany.

PUAUT, I. AND DECOTIGNY, D. 2002. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02: Proceedings of the 23rd IEEE Real-Time Systems Symposium (RTSS'02)*. IEEE Computer Society, Washington, DC, USA, 114.

REINEKE, J. 2008. Caches in WCET analysis. Ph.D. thesis, Universität des Saarlandes.

REINEKE, J. AND GRUND, D. 2008. Relative competitive analysis of cache replacement policies. In *LCTES '08: Proceedings of the 2008 ACM SIGPLAN-SIGBED conference on Languages, compilers, and tools for embedded systems*. ACM, New York, NY, USA, 51–60.

REINEKE, J., GRUND, D., BERG, C., AND WILHELM, R. 2007. Timing predictability of cache replacement policies. *Real-Time Systems 37,* 2, 99–122.

SCHLICKLING, M. AND PISTER, M. 2010. Semi-automatic derivation of timing models for WCET analysis. In *LCTES '10: Proceedings of the ACM SIGPLAN/SIGBED 2010 conference on Languages, compilers, and tools for embedded systems*. ACM, 67–76.

SILBERSCHATZ, A., GALVIN, P. B., AND GAGNE, G. 2005. *Operating System Concepts*. Vol. 2nd. Addison-Wesley.

THEILING, H., FERDINAND, C., AND WILHELM, R. 2000. Fast and precise WCET prediction by separate cache and path analyses. *Real-Time Systems 18,* 2/3.

VERA, X., LISPER, B., AND XUE, J. 2003. Data cache locking for higher program predictability. *SIGMETRICS Perform. Eval. Rev. 31,* 1, 272–282.

WENZEL, I. 2006. Measurement-based timing analysis of superscalar processors. Ph.D. thesis, Technische Universität Wien, Institut für Technische Informatik, Treitlstr. 3/3/182-1, 1040 Vienna, Austria.

WILHELM, R., ENGBLOM, J., ERMEDAHL, A., HOLSTI, N., THESING, S., WHALLEY, D. B., BERNAT, G., FERDINAND, C., HECKMANN, R., MITRA, T., MUELLER, F., PUAUT, I., PUSCHNER, P. P., STASCHULAT, J., AND STENSTRÖM, P. 2008. The worst-case execution-time problem - overview of methods and survey of tools. *ACM Trans. Embedded Comput. Syst. 7,* 3.

WILHELM, R., GRUND, D., REINEKE, J., SCHLICKLING, M., PISTER, M., AND FERDINAND, C. 2009. Memory hierarchies, pipelines, and buses for future architectures in time-critical embedded systems. *IEEE Transactions on CAD of Integrated Circuits and Systems 28,* 7, 966–978.