Sensitivity of Cache Replacement Policies

by

Jan Reineke Daniel Grund

AVACS Technical Report No. 36
March 2008, 1st reprint 2010
ISSN: 1860-9821

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

Sensitivity of Cache Replacement Policies[⋆]

Jan Reineke and Daniel Grund

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

Abstract. Caches are commonly employed to hide the latency gap between memory and the CPU by exploiting locality in memory accesses. On today's architectures a cache miss may cost several hundred CPU cycles. In order to fulfill stringent performance requirements, caches are also used in hard real-time systems. In such systems, upper and lower bounds on the execution time of tasks have to be computed. Different methods have been proposed for timing analysis, including measurement and static analysis.
The sensitivity of a cache replacement policy expresses to what extent the initial state of the cache may influence the number of cache hits and misses during program execution. We have developed a tool to precisely compute sensitivity properties for a large class of replacement policies including LRU, FIFO, PLRU, and MRU. Analysis results demonstrate that the initial state can have a strong impact on the cache performance 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 underestimating the number of misses as strongly as is possible for FIFO, PLRU, and MRU may yield worst-case-execution-time estimates that are far wrong.

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 architectures 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 best- and worst-case execution times (BCET and WCET) of tasks. To obtain tight bounds on

[⋆] This is the first reprint of ATR 36 (with minor modifications), originally titled "Sensitivity Analysis of Cache Replacement Policies". 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 the German-Israeli Foundation (GIF). The second author is supported by a scholarship in the DFG GK 623.

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 initial state of the hardware. A large part of this variation can usually be attributed to the cache performance. At the level of individual instructions, the influence of the initial 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 execution times of multiple instructions cancel each other out, i.e. worst-cases do not coincide [4]. In addition, one may expect different initial states to eventually converge. This is probably true for pipeline states. In many cache architectures, however, this is not the case [3].

Different methods have been proposed for timing analysis [17]; measurement[1] [9,4,16] and static analysis [6,14] 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 states and inputs and derive upper bounds on the execution time of program fragments based on these invariants.
– Measurement executes each program fragment with a subset of the possible initial 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.

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 states and inputs would be covered. Due to their huge number this is usually not feasible. Instead, only a subset of the initial states and inputs can be considered in the measurements. Relatively simple architectures without any performance-enhancing features like pipelines, caches, etc., exhibit the same timing independently of

[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.

the initial state. For such architectures, measurement-based timing analysis is sound [16]. [5] and [16] propose to lock the cache contents [10,15] 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 measurement-based timing analysis can be performed in the presence of "unlocked" caches. To this end, we introduce the notion of sensitivity of a cache replacement policy.

Sensitivity of a cache replacement policy expresses to what extent the initial 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 [12]. 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 are far wrong. In a slightly modified analysis we show that the "empty cache is worst-case initial state" assumption [9] is wrong for FIFO, PLRU, and MRU. 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.

1.1 Outline

The following section reviews some basics about caches. In Section 3 we formally introduce our 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.

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.

Let us briefly explain the three 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.

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 [13,2]. It is used in the PowerPC 75x and the Intel Pentium II-IV.

MRU As described in literature [2].

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

Fig. 1. Domains and Notations

$update_P(q,s) : C^P \times S \to C^P$	function computing the cache-set 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 behaviour of a replacement policy.

3 Sensitivity

In this section, we will formally define our 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 behaviour of a replacement policy. The most important notions are $m_P(q,s)$ and $h_P(q,s)$, which compute the number of 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 *state* on the performance of a cache replacement policy. As cache sets are independent of each other, we consider a single cache set, not the entire cache. I.e. we are interested in how sensitive a policy is to the particular state a cache set is in when beginning to process an access sequence. The results can easily be translated to sensitivity properties of entire caches.

Definition 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-set 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 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-set 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'.

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:

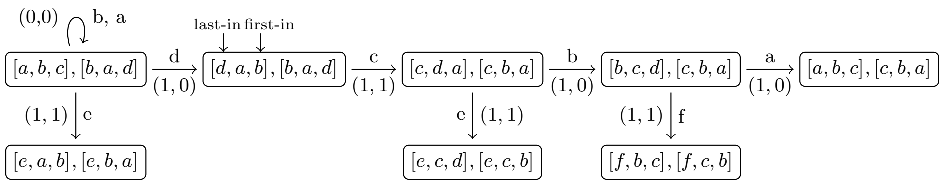


Fig. 3. Running example. Small part of the state space in the computation of sensitivity results for FIFO(3). In the FIFO states, elements are ordered from last-in to first-in. Transitions are labelled with the number of misses incurred. To explain the transitions, we have additionally labelled them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

Definition 3 (Sensitive Ratio). *The sensitive miss and hit ratios s^m_P and s^h_P of P are defined as*

$$s^m_P = \mathit{inf}\ \{k \mid P \text{ is } k\text{-miss-sensitive}\}$$

$$\text{and } s^h_P = \mathit{sup}\ \{k \mid P \text{ is } k\text{-hit-sensitive}\}.$$

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 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 a tool that allows us to compute sensitive ratios automatically. In the following we will describe our approach.

To compute sensitivity values, we construct a transition system, whose states are pairs of cache set states, and whose transitions reflect the effect of a memory access. Figure 3 shows a small part of such a transition system.

The main obstacle is that there are infinitely many cache set states if one assumes the set of memory blocks to be infinite. Although the set of memory blocks is finite in practice, the state space may still be prohibitively large. To overcome this problem we directly 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 (Induced Transition System). *A policy P induces a labelled 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 set states of policy P.

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

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

Sensitivity values depend on the number of misses (hits) on paths through the transition system:

Definition 5 (Paths). *A path through a labelled 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 Set States

We assume cache set states to be tuples of memory blocks and empty cache lines:

$$C_k = \{1, \dots, k\} \to B \cup \{\perp\},$$

where k is the associativity, B is the set of memory blocks and $\perp$ represents empty cache lines. So $C^P \equiv C_k$, where k is the associativity of P. Accesses can have two effects on such a state:

– The order of the memory blocks in the tuple is changed, depending *only* on the *position* of the accessed memory block in the tuple. In FIFO, for instance, the elements may be ordered from last-in to first-in.
– On a miss, an element at a fixed position of the tuple is replaced. In our example of FIFO, this would be the right-most, i.e. the first-in.

Cache set states of all the policies that we consider in this paper, i.e. LRU, PLRU, and FIFO, can be represented this way. However, our approach is not in general limited to policies that allow this kind of representation.

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 sensitivity properties. We have previously employed the same equivalence relation to compute the *competitiveness* of one cache replacement policy *relative* to another policy in [12]. In order to make the paper intelligible in itself, we present a variant of the method presented in [12] which is specifically tailored to the computation of sensitive ratios.

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 set states, that maps $\perp$ (empty cache lines) to $\perp$. 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 set 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 set states are irrelevant for the *possible* future ratio of misses. What is important is the relation between the two cache set states, i.e. the relative positions of elements contained in both sets. Take the two pairs of cache set 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 set states. Yet, we do not want to distinguish the two states, as they will show the same behaviour, albeit on different access sequences.

Let h be as above, and $h^{\#}$ be the point-wise extension of h to pairs of cache set states. We identify two states p and q (each is a pair of cache set 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 $p \approx_h q$.

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

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 labelled 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.

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:

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

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.

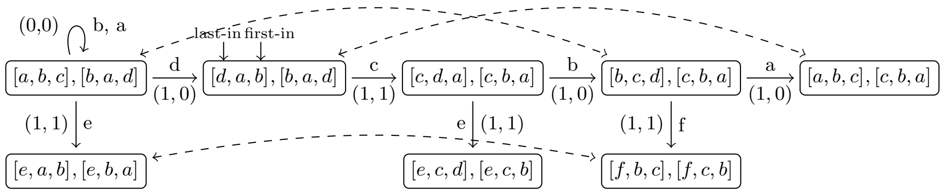
By the following theorem we can safely work with the quotient structure $\overline{T}_P$ instead of T_P when computing sensitivity values.

Theorem 1 (Path Equivalence). *The transition systems T_P and $\overline{T}_P$ are path equivalent, i.e. $\Pi(T_P) = \Pi(\overline{T}_P)$.*

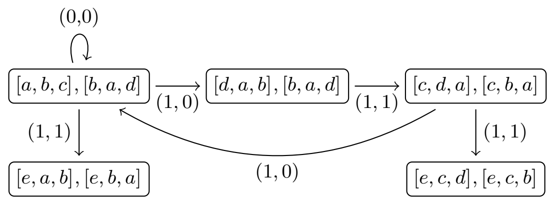
Proof. We need to show $\pi = m_1 \dots m_n \in \Pi(T_P) \Leftrightarrow \pi \in \Pi(\overline{T}_P)$.

"$\Rightarrow$" follows straight-forward 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 $(\overline{s_i}, m_i, \overline{s_{i+1}}) \in \overline{R}_P, \forall i \in \{1,\dots,n\}$:

$$\begin{tikzcd}
 & R_P : & \bullet \arrow[r, "m_1"] \arrow[d, phantom, "\wr"] & \bullet \arrow[r, "m_2"] \arrow[d, phantom, "\wr"] & \bullet \arrow[r, dotted, no head] \arrow[d, phantom, "\wr"] & \bullet \arrow[r, "m_n"] \arrow[d, phantom, "\wr"] & \bullet \arrow[d, phantom, "\wr"] \\
\overset{\text{Def. } \overline{R}_P}{\Rightarrow} & \overline{R}_P : & \circ \arrow[r, "m_1"] & \circ \arrow[r, "m_2"] & \circ \arrow[r, dotted, no head] & \circ \arrow[r, "m_n"] & \circ
\end{tikzcd}$$



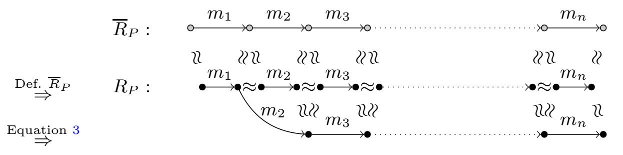
(a) Dashed lines connect equivalent states according to the equivalence relation ≈.



(b) Quotient structure.

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

"⇐" 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:


$\overline{R}_P$:
m_1
m_2
m_3
m_n
R_P :
$\overset{\text{Def. } \overline{R}_P}{\Rightarrow}$
$\overset{\text{Equation 3}}{\Rightarrow}$

□

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

Therefore, the quotient transition system $\overline{T}_P$ is finite. Note that we directly construct $\overline{T}_P$, in particular we never construct the underlying transition system T_P.

4.4 Building 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. $\textsc{Normalize}(p, q)$ computes a unique representative in the equivalence relation for pairs of states. For details on how to compute such a representative see [11].

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 the pair of initial states of the two policies (i^P, i^Q). Instead of computing successors under the same accesses only, we have to take into account arbitrary uncorrelated accesses to both cache-set states. $CC_P(q)$ are the contents of state q, e.g. $CC_{\mathrm{FIFO}(4)}([d, a, b, c]) = \{a, b, c, d\}$. For a more detailed explanation of this part of the algorithm see [11]. $\overline{S}$ denotes the complement of the set S, i.e. $\overline{S} = B \setminus S$, where B is the set of memory blocks.

Once $\overline{S}_P$ has been computed, $\overline{R}_P$ can be computed as follows. The key insight is that $\textsc{Normalize}(update_P(p, \langle a\rangle), update_P(q, \langle a\rangle))$ 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 $CC_P(p) \cup CC_P(q) \cup \{\textsc{SelectOne}(\overline{CC_P(p) \cup CC_P(q)})\}$, where $\textsc{SelectOne}(S)$ selects an arbitrary element of S.

Algorithm 1: Building Quotient Transition System

```
Input: Policy $P$
Output: Quotient Transition System $\overline{T}_P = (\overline{S}_P, \overline{R}_P)$
begin
    $\overline{S}_P \leftarrow \{\textsc{Normalize}(i^P, i^P)\}$
    $Unprocessed \leftarrow [\textsc{Normalize}(i^P, i^P)]$
    while $\neg\textsc{Empty}(Unprocessed)$ do
        $(p, q) \leftarrow \textsc{Pop}(Unprocessed)$
        $m_1 \leftarrow \textsc{SelectOne}(\overline{CC_P(p) \cup CC_P(q)})$
        $m_2 \leftarrow \textsc{SelectOne}(\overline{CC_P(p) \cup CC_P(q) \cup \{m_1\}})$
        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
                $(p', q') \leftarrow \textsc{Normalize}(p', update_P(q, \langle b\rangle))$
                if $(p', q') \notin \overline{S}_P$ then
                    $\textsc{Push}(Unprocessed, (p', q'))$
                    $\overline{S}_P \leftarrow \overline{S}_P \cup \{(p', q')\}$
    $\overline{R}_P \leftarrow \emptyset$
    foreach $(p, q) \in \overline{S}_P$ do
        foreach $a \in CC_P(p) \cup CC_P(q) \cup \{\textsc{SelectOne}(\overline{CC_P(p) \cup CC_P(q)})\}$ do
            $(p', q') \leftarrow \textsc{Normalize}(update_P(p, \langle a\rangle), update_P(q, \langle a\rangle))$
            $(m_p, m_q) \leftarrow (m_P(p, \langle a\rangle), m_P(q, \langle a\rangle))$
            $\overline{R}_P \leftarrow \overline{R}_P \cup \{((p, q), (m_p, m_q), (p', q'))\}$
end
```

4.5 Computation of Sensitive Ratios

Once we have built the quotient transition system, determining the minimal k such that P is hit- (miss-) sensitive amounts to computing the *minimum (maximum) cycle ratio* [8,1]:

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. Miss-sensitivity is a maximum ratio problem. Maximum ratio problems can easily be converted into minimum ratio problems by changing the sign of the numerator or the denominator.

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

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

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

[8,1] describe how to solve the minimum cycle ratio problem by repeated applications of a negative cycle detection algorithm.

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

Theorem 3 (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

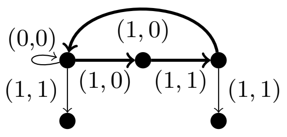
$$\sum_i \pi(i)_{|1} \leq k \cdot \sum_i \pi(i)_{|2} + c \text{ for all paths } \pi \in \mathit{\Pi}(T_P).$$

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 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 $c \leq 2 \cdot |\overline{S}_P|$.

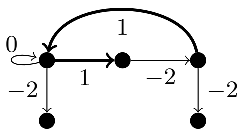
For 2.: Choose a "cyclic" path π that corresponds to the maximal cycle ratio k. As $k > k'$, $\sum_i \pi(i)_{|1} = k \cdot \sum_i \pi(i)_{|2} > k' \cdot \sum_i \pi(i)_{|2}$. By repeating π appropriately often $\sum_i \pi^n(i)_{|1} > k' \cdot \sum_i \pi^n(i)_{|2} + c'$ for any additive constant c'. □

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, which is not explained by the sensitive ratio 2. 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.



(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.

5 Results

Using our tool, we have obtained sensitivity results for LRU, FIFO, and PLRU 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 and PLRU are much more sensitive to their state than LRU.

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

	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-Sensitivity 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-Sensitivity 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. PLRU is only defined for powers of two. MEM indicates that the algorithm ran out of memory on a 2 GB machine.

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: no lower bound on the number of hits can be given for one state, given 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 cause only 1/11 of the number of hits depending on the starting state. See Figure 7(b) for the analysis results.

Summarizing, both FIFO and PLRU may in the worst-case be heavily influenced by the starting state. LRU is very robust, in that the number of hits and misses is affected in the least possible way. A study of predictability of the above policies [13] provides limits on the precision of static cache analyses: While static analysis yields correct results, it will usually not be precise for FIFO and PLRU. In contrast, it can be very precise for LRU.

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

One could argue that it is still safe to assume an empty cache[2] as the starting state [9, page 39ff], assuming that an empty cache were worse than any non-empty cache. This is *not* true for FIFO 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. It has been observed earlier [3], 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 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 [7]. 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}:

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

[2] or equivalently a cache filled with irrelevant data only

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? Let the number of memory accesses per instruction be 1.2[3] and let the miss penalty be 50. Due to pipeline stalls let the CPI_{hit} be 1.5. Further assume, the miss rate Miss rate_{meas} during the measurement was 5%. 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 20%. Plugging the above assumptions into the equations and simplification yields

$$\begin{aligned}\frac{T_{wc}}{T_{meas}} &= \frac{\text{CPI}_{hit} + \frac{\text{Mem. accesses}}{\text{Instruction}} \times \text{Miss rate}_{wc} \times \text{Miss Pen.}}{\text{CPI}_{hit} + \frac{\text{Mem. accesses}}{\text{Instruction}} \times \text{Miss rate}_{meas} \times \text{Miss Pen.}}\\ &= \frac{1.5 + 1.2 \times 0.20 \times 50}{1.5 + 1.2 \times 0.05 \times 50} = \frac{13.5}{4.5} = 3\end{aligned}$$

So, in the example of a 4-way set-associative FIFO cache, the worst-case execution time may be a factor of 3 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 [9,4,16,5] 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 initial state of the cache on the future cache performance. Employing techniques first described in [12] allows to compute sensitive

[3] Each instruction causes one instruction fetch and possibly data fetches.

ratios of a large class of replacement policies, including the three well-known and widely-used families of replacement policies, LRU, FIFO, and PLRU.

The analysis results revealed great differences among LRU, FIFO, and PLRU, that yield another argument in favour of using LRU in the design of predictable real-time systems. In the case of FIFO and PLRU, the initial state can have a great influence on the number of hits and misses during program execution. A simple model of execution time demonstrates the impact of cache sensitivity on measured execution times. It shows that underestimating the number of misses as strongly as is possible for FIFO and PLRU yields worst-case-execution-time estimates that are far wrong. Further analysis revealed that the "empty cache is worst-case initial state" assumption [9] is wrong for FIFO and PLRU.

To obtain safe results by measurement with respect to cache performance the cache contents should be locked as proposed in [5,16,15,10], which may have an adverse effect on average- and worst-case execution time. Our results hold for arbitrary access sequences. Therefore, they hold for any program. Many programs do not exhibit the worst-case cache behaviour. 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.

References

1. R. K. Ahuja, T. L. Magnanti, and J. B. Orlin. *Network flows: theory, algorithms, and applications*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1993.
2. H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *ACM-SE 42*, pages 267–272, New York, NY, USA, 2004.
3. C. Berg. PLRU cache domino effects. In F. Mueller, editor, *WCET '06*, 2006.
4. G. Bernat, A. Colin, and S. M. Petters. WCET analysis of probabilistic hard real-time systems. In *RTSS '02*, page 279, Washington, DC, USA, 2002.
5. J.-F. Deverge and I. Puaut. Safe measurement-based wcet estimation. In R. Wilhelm, editor, *WCET '05*, Dagstuhl, Germany, 2005.
6. C. Ferdinand, R. Heckmann, M. Langenbach, F. Martin, M. Schmidt, H. Theiling, S. Thesing, and R. Wilhelm. Reliable and precise WCET determination for a real-life processor. In *Embedded Software Workshop*, volume 2211, pages 469 – 485, October 2001.
7. J. L. Hennessy and D. A. Patterson. *Computer Architecture: A Quantitative Approach, 3rd Edition*. Morgan Kaufmann, 2003.
8. E. Lawler. Optimal cycles in doubly weighted linear graphs. In *Int'l Symp. Theory of Graphs*, 1966.
9. S. M. Petters. *Worst Case Execution Time Estimation for Advanced Processor Architectures*. PhD thesis, Technische Universität München, Sept. 2002.
10. I. Puaut and D. Decotigny. Low-complexity algorithms for static cache locking in multitasking hard real-time systems. In *RTSS '02*, USA, 2002.
11. J. Reineke. *Caches in WCET Analysis*. PhD thesis, Saarland University, Saarbrücken, November 2008.

12. J. Reineke and D. Grund. Relative competitive analysis of cache replacement policies. In *LCTES'08*, 2008.
13. J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, November 2007.
14. H. Theiling, C. Ferdinand, and R. Wilhelm. Fast and precise WCET prediction by separate cache and path analyses. *Real-Time Systems*, 18(2/3), May 2000.
15. X. Vera, B. Lisper, and J. Xue. Data cache locking for higher program predictability. *SIGMETRICS Perform. Eval. Rev.*, 31(1):272–282, 2003.
16. I. Wenzel. *Measurement-Based Timing Analysis of Superscalar Processors.* PhD thesis, Technische Universität Wien, Institut für Technische Informatik, 2006.
17. R. Wilhelm, J. Engblom, A. Ermedahl, N. Holsti, S. Thesing, D. Whalley, G. Bernat, C. Ferdinand, R. Heckmann, T. Mitra, F. Mueller, I. Puaut, P. Puschner, J. Staschulat, and P. Stenström. The worst-case execution-time problem—overview of methods and survey of tools. *Transactions on Embedded Computing Systems*, 7(3):1–53, 2008.