Estimating the Performance of Cache Replacement Policies*

Daniel Grund and Jan Reineke
Saarland University
{grund,reineke}@cs.uni-sb.de

Abstract

Caches are commonly employed to hide the latency gap between memory and the CPU by exploiting locality in memory accesses. The cache performance strongly influences a system's overall performance, as this gap is large and ever-increasing. The efficiency of a given cache architecture – usually measured by its miss ratio – varies greatly depending on the software being executed.

We present an efficient method to estimate the miss ratio using a stochastic model. The model takes into account the parameters of the cache architecture and a concise characterization of the software's locality.

In contrast to previous approaches, we consider the replacement policy as an important component of the cache architecture. To this end, we introduce policy tables as a concise representation of replacement policies. The software's locality is characterized by stack histograms or our extension thereof: History stack histograms, which refine stack histograms by distinguishing contexts of accesses.

Simulation results on the SPEC benchmarks demonstrate the strong influence of the replacement policy on the miss ratio and the precision of our estimates: average absolute errors between 0.18% *and* 2.92%.

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, handling a cache miss may take several hundred CPU cycles. Future architectures are expected to exhibit even larger cache-miss penalties. Therefore, the cache performance – usually measured by its miss ratio – has an increasingly strong influence on a system's overall performance.

Accurate prediction of miss ratios is important in memory architecture design [7], compiler optimizations [8, 10, 11], and the hardware selection process for a given software system. Of course, miss ratios vary greatly depending on the program (code and input) being executed. In particular, there is not a single ordering of cache architectures (according to miss ratio) that holds for all programs. Instead, every program induces its own ordering of cache architectures regarding the incurred miss ratio.

Obtaining miss ratios by simulation is very time consuming and therefore too expensive or impractical in many cases. We present an efficient analytical method to estimate miss ratios for a given cache configuration. Our method requires a concise characterization of the program's locality by well-known stack histograms [14] or extensions thereof. We take into account the most important parameters of cache architectures:

Line Size l: The size of memory blocks stored in the cache.
Associativity k: The number of lines in the cache that one particular memory block may be stored in.
Number of Sets s: The number of cache sets, where each cache set consists of k cache lines.
Capacity: The overall capacity $C = s \cdot k \cdot l$ of the cache.
Replacement Policy: The function that determines the cache line whose contents are replaced upon a cache miss.

The replacement policy can have a significant influence on the cache performance [1]. The previous approaches [14, 8, 9, 5] for estimating miss ratios assumed a fixed replacement policy (LRU or random). Recent work [12] considers some policies. However, besides LRU, they only consider non-classical, virtual policies that can be directly specified in form of so-called replacement probability functions (RPF). Instead we make the policy a first-class citizen. To this end, we introduce a uniform representation of a large class of policies including the most prominent ones: LRU, MRU, PLRU, and FIFO. This representation captures the behavior of a policy concisely and uniformly, which eases the task of building the prediction model. Furthermore, we extend the well-known stack histograms with history information to improve the precision of estimates.

We explicitly discuss all sources of inaccuracy: the ones inherent to the model input as well as those introduced by the model to increase efficiency. We use SPEC CPU2000 benchmarks and compare our estimated miss ratios to ones obtained by costly simulation. For the considered policies

*This work has profited from discussions within the ARTIST2 Network of Excellence. It was supported by the German Research Foundation.

the average absolute prediction errors vary between 0.18% and 2.92%.

In the next section, we explain stack histograms and introduce an extension: history stack histograms. The first part of Section 4 introduces a uniform representation of replacement policies and the other parts focus on our stochastic Markov chain model. The evaluation results are contained in Section 5. Finally, Section 6 contrasts this paper to related work.

2. Stack Histograms

In 1970, Mattson et al. [14] introduced the notions of *inclusion*, *stack distance*, and *stack histograms* and derived a technique called *stack processing*. Stack histograms concisely capture locality in the memory-access behavior of a program. They are defined using the following notions.

- The *age* of a cacheable element e, $age(e) \in \mathbb{N}_0$, is the number of different elements that map to the same set and have been accessed after the last access to e. The relations older and younger are defined in the obvious way. Note, that elements may age (get older) even if they have already been evicted from the cache.
- The *stack distance* d (also called reuse distance) of a memory access is the age of the accessed element just before that access. The first reference to an element has distance $d = \infty$.
- The *stack histogram* of a program run is the frequency distribution $f_0, f_1, \ldots$ of the stack distances that occur during execution.

In practice one limits the number of bins in the distribution. Stack distances larger than a certain threshold are summed up in a final bin as illustrated in the following example. The collection process of data would look like this:

Reference string	a	b	a	c	b	b	c	a
Stack distance	∞	∞	1	∞	2	0	1	2

And the resulting stack histogram could be represented as:

Stack distance	0	1	2	3	≥ 4
Rel. frequency	$\frac{1}{8}$	$\frac{2}{8}$	$\frac{2}{8}$	0	$\frac{3}{8}$

Caches exploit locality by storing memory contents expected to be referenced in the near future. To estimate the miss ratio of a program-run on a certain cache architecture, an analysis can combine the empirical information provided by a stack histogram with static knowledge about the functioning of a cache. Consider the following example of a cache using LRU replacement. Since LRU always retains the k most recently used elements, exactly those elements having an age $< k$ are cached. Thus, given a stack histogram $f_0, f_1, \ldots$ the miss ratio of a k-way associative cache can simply be determined by $\text{miss ratio} = \sum_{d=k}^{\infty} f_d = 1 - \sum_{d=0}^{k-1} f_d$. This formula is exact for LRU. For other replacement policies the formula may be inaccurate, since, in general, elements older than k may be cached. In the case of PLRU the age of a cached element is even unbounded, i.e. provided a sufficient number of accesses to different elements, arbitrarily old elements may be cached [4].

To obtain an even more compact representation of a program's access behavior, one may summarize access frequencies to age intervals [16, 10]. To cover associativities that are powers of two these intervals are usually chosen as 1-1, 2-2, 3-4, 5-8, 9-16, etc. Although this works well for prediction of miss ratios in LRU caches this representation is not useful for our purpose.

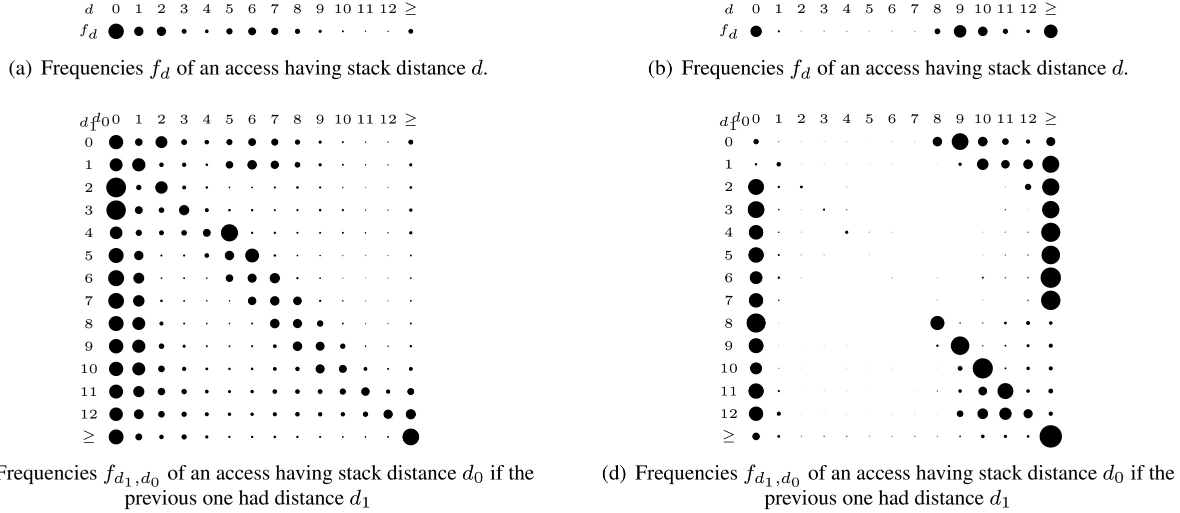
Note, that stack histograms depend on the number of sets and the line size of a cache. Both values determine which elements are mapped together to one set, and additionally the line size determines to what extent an access to a neighboring address in fact accesses the same (atomic) element. On the other hand, stack histograms neither depend on the associativity nor on the replacement policy. Raising the associativity while keeping the number of sets constant (and thereby raising the overall size of the cache) results in the same stack histogram. Thus, one stack histogram covers a family of cache configurations.

Stack histograms can be determined by simulation or by processing a hardware-captured address trace. The dependency on the number of sets and the line size can be overcome in two ways. First, one can expand the simulation run to handle multiple families of caches simultaneously [13] and use our model to obtain predictions for the remaining free dimensions within each family. Second, it is possible to convert stack histograms. Hill and Smith [13, 19] give an approximation formula to convert a stack histogram of a fully associative cache into one for a cache with more sets.

2.1. Accuracy of Stack Histograms

Stack histograms offer a very good trade-off between accuracy and storage/processing costs. Storing whole address traces would require tremendous amounts of disk space, e.g., the SPEC benchmark `164.gzip` with the `program` input results in approximately 28 billion data cache accesses. The price that comes along with the concise representation as stack histogram is a certain loss in precision.

Obviously, one may lose some information by restricting the number of bins. But even with infinitely many bins a stack histogram cannot reflect a program's memory access behavior perfectly. One disregards the following facts when storing one stack histogram per cache per program: First, the program may stress different cache sets differently. Sec-



(a) Frequencies f_d of an access having stack distance d.

(b) Frequencies f_d of an access having stack distance d.

(c) Frequencies f_{d_1,d_0} of an access having stack distance d_0 if the previous one had distance d_1

(d) Frequencies f_{d_1,d_0} of an access having stack distance d_0 if the previous one had distance d_1

Figure 1. Circle plots of stack histograms with different history lengths. The areas of circles are proportional to frequencies. The histograms were recorded for a shared 2nd-level cache while simulating `176.gcc` with the `integrate` input. a) Without history one can only see that accesses on younger elements are more likely. c) History length `1` provides more information: the bold diagonal shows that after accessing an older element an access to an element of the same or a similar age is more likely than suggested by the non-history stack histogram. b,d) Example of an unusual distribution: `179.art` with `ref.2` input.

ond, the program may change its memory access behavior over execution time.

In other words, errors may be introduced by taking the average over all cache sets and over time. The first source of inaccuracy can be completely eliminated if one is willing to spend additional space to store one stack histogram per cache set. But this is unlikely to allow for great improvements in the precision of predicted miss ratios [1]. The second source has a greater impact on the quality of predictions, but is more difficult to eliminate. One possibility is to detect significant changes in the program behavior [18, 15] during simulation and record one stack histogram per program phase. Such methods could be combined with our estimation approach to further improve its precision.

2.2. History Stack Histograms

To increase precision we propose another orthogonal improvement. Independent of the number of stack histograms used to characterize a program, we propose to enrich stack histograms. Instead of considering stack distances of accesses in isolation, one can also take into account their context. We extend stack histograms by choosing the context to be stack distances of previously accessed elements (in the same set). See Figure 1 for examples.

Definition 1. *An* l-history stack histogram *is a set of frequencies $f_{d_l,\ldots,d_1,d_0}$. l is the history length and $f_{d_l,\ldots,d_1,d_0}$ is the frequency of stack distance d_0 given that the previous accesses had stack distances $d_l,\ldots,d_1$ (d_1 being the distance of the most recent access).*

The distinction of different sequences of consecutive accesses by this context sensitivity should in principle map to the distinction of different usage patterns of a cache set. Increasing history length arbitrarily would provide the full information contained in the corresponding address trace. Whether large history lengths are sufficient to distinguish different program phases, and whether these are easy-to-handle in an efficient way, is unclear and beyond the scope of this paper.

In Section 5 we will demonstrate that incorporating history improves the precision of estimated miss ratios.

3. Representing Policies by Policy Tables

The class of replacement policies we consider is restricted to policies that treat all cache sets uniformly and independently: all cache sets behave the same, and accesses in one cache set do not influence other cache sets. Furthermore, replacement decisions of a set are based solely on an order of its cached elements. This order is only virtually imposed, i.e. it need not be explicitly maintained in the

$$\begin{bmatrix} \pi_0 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_1 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_2 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_3 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_4 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_5 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_6 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_7 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_m & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 0 \end{bmatrix}$$

(a) FIFO

$$\begin{bmatrix} \pi_0 & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 0 \\ \pi_1 & 0\ 2\ 3\ 4\ 5\ 6\ 7\ 1 \\ \pi_2 & 0\ 1\ 3\ 4\ 5\ 6\ 7\ 2 \\ \pi_3 & 0\ 1\ 2\ 4\ 5\ 6\ 7\ 3 \\ \pi_4 & 0\ 1\ 2\ 3\ 5\ 6\ 7\ 4 \\ \pi_5 & 0\ 1\ 2\ 3\ 4\ 6\ 7\ 5 \\ \pi_6 & 0\ 1\ 2\ 3\ 4\ 5\ 7\ 6 \\ \pi_7 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_m & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 0 \end{bmatrix}$$

(b) LRU

$$\begin{bmatrix} \pi_0 & 4\ 5\ 6\ 7\ 2\ 3\ 1\ 0 \\ \pi_1 & 4\ 5\ 6\ 7\ 2\ 3\ 0\ 1 \\ \pi_2 & 4\ 5\ 6\ 7\ 0\ 1\ 3\ 2 \\ \pi_3 & 4\ 5\ 6\ 7\ 0\ 1\ 2\ 3 \\ \pi_4 & 0\ 1\ 2\ 3\ 6\ 7\ 5\ 4 \\ \pi_5 & 0\ 1\ 2\ 3\ 6\ 7\ 4\ 5 \\ \pi_6 & 0\ 1\ 2\ 3\ 4\ 5\ 7\ 6 \\ \pi_7 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_m & 4\ 5\ 6\ 7\ 2\ 3\ 1\ 0 \end{bmatrix}$$

(c) PLRU

$$\begin{bmatrix} \pi_0 & 0\ 1\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_1 & 1\ 0\ 2\ 3\ 4\ 5\ 6\ 7 \\ \pi_2 & 2\ 0\ 1\ 3\ 4\ 5\ 6\ 7 \\ \pi_3 & 3\ 0\ 1\ 2\ 4\ 5\ 6\ 7 \\ \pi_4 & 4\ 0\ 1\ 2\ 3\ 5\ 6\ 7 \\ \pi_5 & 5\ 0\ 1\ 2\ 3\ 4\ 6\ 7 \\ \pi_6 & 6\ 0\ 1\ 2\ 3\ 4\ 5\ 7 \\ \pi_7 & 7\ 0\ 1\ 2\ 3\ 4\ 5\ 6 \\ \pi_m & 1\ 2\ 3\ 4\ 5\ 6\ 7\ 0 \end{bmatrix}$$

(d) MRU

$$\begin{bmatrix} \pi_0 & 1\ 4\ 2\ 5\ 6\ 3\ 0\ 7 \\ \pi_1 & 5\ 2\ 6\ 3\ 4\ 1\ 7\ 0 \\ \pi_2 & 0\ 2\ 3\ 5\ 1\ 7\ 6\ 4 \\ \pi_3 & 4\ 1\ 6\ 3\ 0\ 2\ 7\ 5 \\ \pi_4 & 6\ 4\ 3\ 1\ 2\ 5\ 7\ 0 \\ \pi_5 & 2\ 4\ 0\ 3\ 7\ 6\ 1\ 5 \\ \pi_6 & 4\ 0\ 3\ 5\ 2\ 1\ 6\ 7 \\ \pi_7 & 0\ 5\ 6\ 2\ 4\ 3\ 1\ 7 \\ \pi_m & 1\ 2\ 7\ 0\ 6\ 3\ 4\ 5 \end{bmatrix}$$

(e) 8-way RAND

$$\begin{bmatrix} \pi_0 & 2\ 1\ 0\ 3 \\ \pi_1 & 2\ 0\ 1\ 3 \\ \pi_2 & 2\ 0\ 3\ 1 \\ \pi_3 & 1\ 0\ 2\ 3 \\ \pi_m & 3\ 0\ 1\ 2 \end{bmatrix}$$

(f) 4-way RAND

Figure 2. Policy tables of policies considered in this paper. Since RAND has no systematic construction we show tables for both associativities.

hardware implementation. For example, in LRU the elements are ordered from least- to most-recently used. Upon an access, this order is updated based on the position of the accessed element in the order. This implies that – besides distinguishing hits from misses – tags and cached data do not influence replacement behavior. In case of LRU the order is updated by moving the accessed element to the most-recently used position. In principle, our prediction model could deal with other representations of policies as well, but as one will see next, policy tables are a convenient way to express the functioning of policies in a uniform way. This view of policies allows for the following formalization.

Definition 2. *A* policy state *is a permutation π of the cache lines $\{0, \dots, k-1\}$ in a cache set. We will speak of $\pi(p)$ as the line at* position p.

Definition 3. *The* eviction position *is (w.l.o.g.) defined to be position 0. Upon a cache miss the element in the cache line at the eviction position 0 will be replaced.*

In case of LRU the least-recently used position has to be position 0. Note that no statement is given about the order in which other elements are moved to the eviction position. I.e. the element at position 1 is not necessarily the next to be replaced.

Now let us introduce a uniform representation for the policies in the considered class:

Definition 4. *An* policy table $T = [\pi_0, \dots, \pi_{k-1}, \pi_m]$ *is a tuple of $k+1$ permutations over the set $\{0, \dots, k-1\}$. It specifies a replacement policy for a k-way set-associative cache. Upon a cache hit on the line at position p in a cache-set state π, the successor state is $\pi' = \pi \circ \pi_p$. Upon a cache miss, $\pi' = \pi \circ \pi_m$.*

Note, that $a \circ b$ means first b then a. We mean to say that the π_i do not specify where positions are mapped to $(\pi'(p) = \pi_i(\pi(p)))$, instead they specify positions to take from $(\pi'(p) = \pi(\pi_i(p)))$.

Note, that policy tables are not unique, i.e., there are different tables that exhibit the same replacement behavior. Of $k!^{k+1}$ lexically different policy tables at least $k!^k$ show different replacement behavior.

To become familiar with policy tables have a look at the policy tables of the well-known policies in Figure 2.

First, consider FIFO (first-in first-out). Since hits do not influence replacement decisions, all hit-permutations $\pi_0, \dots, \pi_7$ are the identity permutation. Only a miss can change a cache-set state. By definition the element in the cache line at position 0 is replaced and then the update $\pi' = \pi \circ \pi_m$ is performed. pi_m assigns the just filled cache line the position 7 and the positions of the other cache lines are decreased by one (moved to the left). That is, cache lines are ordered from first-in to last-in from position 0 to 7.

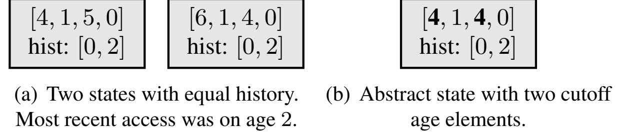
In LRU (least-recently used) replacement the miss permutation is the same but hits have an effect, too. Notice the "triangular" structure: the accessed cache line is always moved to position 7 ($\pi_i(7) = i$), lines at positions greater than the accessed one are moved to the left, and the remaining lines are not affected. Thus, the cache lines are ordered from least to most recently used (i.e. from old to young) from position 0 to 7.

Explaining the construction of the PLRU (pseudo LRU) policy table is not that easy. But one can see that the tree-structure of PLRU is reflected in the table by "blocks" of size 4, 2, and 1. For a general explanation how PLRU works see e.g. [1, 17].

Most recently used (MRU) is a counterpart to LRU: The miss permutation is the same as in LRU, but a hit on an element moves that element to the replacement position 0. Only a cache miss places the newly inserted element at the right-most position, saving it from immediate replacement.

The RAND-policies were created by choosing a random permutation for each line in the policy table. Contrary to

the other policies, the RAND tables have no systematic construction pattern. Thus, there is no relation between RAND policies of different associativities. Note, that random does not mean that every cache line is replaced with probability $\frac{1}{k}$. Truly random behavior cannot be modeled by policy tables as they are deterministic. Nonetheless they are pseudo-random and we consider them to see how our model behaves with unstructured input. Our general approach would however allow to estimate miss rates for randomized policies as well.



(a) Two states with equal history. Most recent access was on age 2.

(b) Abstract state with two cutoff age elements.

Figure 3. Examples for Markov chain states. With cutoff age 4 the two states from (a) are subsumed by the abstract state shown in (b).

4. Stochastic Model

In this section we explain how to compute estimates of miss ratios given a stack histogram and a replacement policy. We build a Markov chain based on the following inputs:

- a history stack histogram
- a policy table
- a cutoff age, which is a parameter to keep the model finite and is explained later in detail

The stack histogram being used determines the line size and the number of sets for which the computed estimates hold. In addition, the policy table specifies replacement policy and associativity.

4.1. Motivation

As noted before in Section 2, given a stack histogram it is easy to compute the miss ratio for LRU replacement. LRU retains the k most recently used elements, i.e. elements of age $0, \ldots, k-1$. As a stack histogram specifies the access frequencies for those ages, the miss ratio is $1 - \sum_{d=0}^{k-1} f_d$. For other policies, estimation is more difficult because the ages of cached elements vary during execution, i.e. cache sets do not always contain elements of ages $0, \ldots, k-1$. The number and shape of possible combinations of ages in a cache set strongly depends on the policy.

Definition 5. *A* cache-set state *is a tuple of ages* $[a_0, \ldots, a_{k-1}]$ *that defines a mapping from positions (as defined in the previous section) to ages (*not *from cache lines to ages). It abstracts form the physical arrangement in cache lines by incorporating the policy state, i.e. it only matters in which order the elements are replaced.*

The idea to compute a miss ratio estimate is to compute for all cache-set states:

- the probability that a miss happens in that state
- the probability that a cache set is in that state

Then, the miss ratio estimate is simply the inner product of these two vectors of probabilities.

The miss probability of a given cache-set state is easy to compute using the stack histogram: one sums up the frequencies of ages that are absent in the cache-set state.

The probability that a cache set is in a certain state *at a certain point in time* depends on the probabilities of predecessor states and the respective transition probabilities. As time passes the influence of the initial probability distribution vanishes; the system assumes a steady state. This process can be modeled as a Markov chain. Then, the wanted probability of a cache set being in a particular state is simply the steady state probability of that state in the Markov chain. The following subsection describes how to build the Markov chain.

4.2. Model Construction

In our Markov chain, a state $s = (a, h)$ comprises a *cache-set state* a and *history* h. The cache-set state represents the ages contained in the set as defined above. The history is a tuple of stack distances $[d_l, \ldots, d_1]$ of the most recently accessed elements. It is used to select the correct probabilities in the history stack histogram (note that from now on we will interpret the measured frequencies as probabilities, i.e. $p_* := f_*$). See Figure 3(a) for examples. Transitions in the Markov chain model the effect of a memory access on the cache-set state and on the history.

In such a model, arbitrarily high ages are a problem. Firstly, stack histograms are finite and cannot provide frequencies for arbitrarily high stack distances. Secondly, as mentioned in Section 2, elements with arbitrarily high ages may be cached. Hence, the state space may be infinite (or prohibitively large). To overcome this problem we use the *cutoff age* parameter mentioned above. The probability of accessing an element usually decreases strongly with its age. Thus, we abstract from high ages by subsuming into the cutoff age all ages greater than or equal to this threshold. The cutoff age will be denoted by bold numbers. See Figure 3(b) for an example.

In the following, we describe the construction of the model, i.e. its state space and transitions. First we need

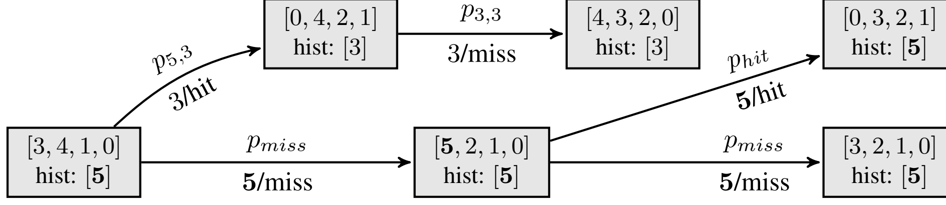


Figure 4. Example (FIFO, history length 1, cutoff age 5) showing some transitions and how probabilities are assigned to them. The value above an arrow is the transition probability. p_{hit} and p_{miss} are short for $p_{hit}(a,h)$ and $p_{miss}(a,h)$, respectively. The pair below an arrow shows the age that is accessed and whether this access is a hit or a miss. Note, that in state $[\mathbf{5},2,1,0]$ an access to an element with cutoff age (≥ 5) may be a hit or a miss. Thus, the "hit" or "miss" annotations below the arrows are assumptions, not consequences.

to introduce some notation. States $s \in S = A \times H$ will be written as $s = (a,h)$ with cache-set state $a = [a_0, \ldots, a_{k-1}]$ and history $h = [d_l, \ldots, d_1]$. The input required by our model is given in the following form:

- A history stack histogram $HSH = (p_{d_l,\ldots,d_1,d_0})$
- A policy table $T = [\pi_0, \ldots, \pi_{k-1}, \pi_m]$
- A cutoff age c

During state space exploration, for each state $s = (a,h)$, we enumerate all possible accesses to generate all its successor states. Below we list all cases in form of (a', h', p), where $s' = (a', h')$ is the successor state and p is the probability of the transition (s, s').

1. An access having stack distance $d < c$. These are the simple cases: First, the distinction between hits and misses is made. In case of a miss the function $repl : A \times \mathbb{N} \rightarrow A$ replaces the element at position 0 with an element of a given age.

   $$repl([a_0, a_1, \ldots, a_{k-1}], a) := [a, a_1, \ldots, a_{k-1}]$$

   Second, aging is performed. The function $age : A \times \{0, \ldots, k-1\} \rightarrow A$ updates the ages of a state depending on the accessed position. According to the definition of age, all elements younger than the accessed one age by one. The accessed element obtains age 0.

   $$age([a_0, \ldots, a_{k-1}], pos) := [a'_0, \ldots, a'_{k-1}], \text{ where}$$

   $$a'_j = \begin{cases} 0 & \text{if } j = pos \\ \min\{c, a_j + 1\} & \text{if } j \neq pos, a_j < a_{pos} \\ a_j & \text{otherwise} \end{cases}$$

   Third, positions are updated according to the policy by applying a permutation. Finally, the history is adjusted, and the probability is directly taken from the history stack histogram. The two accesses at the top of Figure 4 are of this kind. In summary:

   $$\begin{array}{rcll} a' & = & \begin{cases} age(a,i) \circ \pi_i & \exists i : a_i = d \quad \text{(hit)} \\ age(repl(a,d),0) \circ \pi_m & \forall i : a_i \neq d \quad \text{(miss)} \end{cases} \\ h' & = & [d_{l-1}, \ldots, d_1, d] \\ p & = & p_{d_l,\ldots,d_1,d} \end{array}$$

2. A hit at position i having stack distance $d = c$, e.g. the "**5**/hit" transition in Figure 4. Since one does not know the exact age of the element it is unclear which probability to assign. Assuming that the probability to replace an element of age $\geq c$ upon a miss is $\frac{1}{k}$, the probability that a cutoff age element has age i is:

   $$p_{age}(i) := \begin{cases} 0 & : \quad i < c \\ \frac{1}{\lambda}(1 - \frac{1}{k})^{i-c} & : \quad i \geq c \end{cases}$$

   where λ ensures $\sum_{i=0}^{\infty} p_{age}(i) = 1$. Thus,

   $$\lambda = \sum_{i=0}^{\infty} \left(1 - \frac{1}{k}\right)^i = \frac{1}{1 - (1 - \frac{1}{k})} = k.$$

   Then, the probability of accessing an element with cutoff age is:

   $$p_{hit}(a,h) := \sum_{i=c}^{\infty} p_{age}(i) \cdot p_{h,i}$$

   If there is more than one cutoff age element, in very rare cases the sum of the assigned access probabilities (cases 1 and 2) may exceed 1. In such cases we reduce $p_{hit}(a,h)$ such that the sum equals 1.

   Finally,

   $$\begin{array}{rcl} a' & = & age(a,i) \circ \pi_i \\ h' & = & [d_{l-1}, \ldots, d_1, c] \\ p & = & p_{hit}(a,h) \end{array}$$

3. A miss having stack distance $d = c$, e.g. the two "**5**/miss" transitions in Figure 4. This includes compulsory misses. Since this is the last possible case it gets assigned the remaining outgoing probability:

$$p_{miss}(a,h) := 1 - \underbrace{\sum_{i=0}^{c-1} p_{h,i}}_{\text{case 1}} - \underbrace{cc(a) \cdot p_{hit}(a,h)}_{\text{case 2}}$$

where the function $cc : A \to \mathbb{N}$ counts the number of cutoff age elements in a cache-set state.

$$\begin{aligned} a' &= age(repl(a,c),0) \circ \pi_m \\ h' &= [d_{l-1}, \ldots, d_1, c] \\ p &= p_{miss}(a,h) \end{aligned}$$

When constructing the chain, we need some state to start state space exploration from. Since (partially) empty cache-set states are not contained in our model we have to start from a completely filled state. Simply choosing some configuration of ages (and history) is not possible because such a state might be infeasible, i.e. impossible to reach from the empty initial state. To construct such a state we assume that the given policy table satisfies one (very weak) condition: Starting from the empty initial cache-set state some state must be reachable in which all cache lines are filled, i.e. the cache can be fully utilized. Any reasonable policy satisfies this condition, otherwise it would waste parts of the cache. In our implementation we construct such a completely filled state and start model construction from this state.

Not considering (partially) empty cache-set states is not a problem: They would not be part of a terminal strongly connected component. Thus, they would always have steady-state probability 0 and not contribute to the estimate.

4.3. Deriving the Miss-Ratio Estimate

As mentioned above the final estimate is obtained by an inner product of two probability vectors. The probabilities m for cache misses in each cache-set state and the probabilities p for being in a cache-set state. m is simply determined by the cache-set states and the stack histogram. p is given by the steady state probabilities of the Markov chain. To obtain p one has to solve the eigenvalue problem $Tp = p$, where T is the transition probability matrix of the Markov chain. We use the Gauss-Seidel Method [3] to solve this subproblem. Then, the final miss-ratio estimate is $p^T m$.

This approach is based on the assumption that the Markov chain assumes a steady state, i.e. as time passes the influence of the initial state of the cache vanishes. The following theorem shows that this assumption holds by proving a sufficient condition.

Theorem 1. *The finite Markov chain we build is irreducible and aperiodic.*

Proof. Irreducibility: For any two states r and s we have to show that s is reachable from r. By premise, s is reachable from the empty initial state by some access sequence a_s. Let $F_s = \{e \in a_s\}$ be the set of all elements in such an access sequence a_s. Since all cache sets can be fully utilized one can also replace all elements in a cache set. Thus starting from r, a state t is reachable that does not contain any elements of F_s. Since t has no elements in common with the access sequence a_s, the contents of t cannot influence the execution of a_s. With respect to execution of a_s the empty initial cache-set state and t are equivalent. Hence, s is reachable from t by applying a_s. This reasoning about cache-set states can be directly transferred to the (abstract) Markov chain states.

Aperiodicity: Let n be larger than all of cutoff age, associativity, and history length ($n > c, k, l$). Additionally let $c \geq k$. Let $P := \{p \mid p = \pi_m^i(0)\}$ be the set of positions that are evictable by π_m.

Consider a Markov chain state t and an access sequence of length n such that each accessed element has cutoff age and all accesses are cache misses. Let the resulting state be $s = ([a_i], h)$. Since $n > k$ and only π_m was applied all ages a_i on evictable positions have ages $0, \ldots, |P| - 1$. Furthermore they are ordered according to the cycle of π_m that contains 0. Additionally, since $n > c$ all elements on other positions have cutoff age. Finally, since $n > h$ and only cutoff-age elements were accessed $h = [c, \ldots, c]$. Then, one additional access of the same kind corresponds to the transition $s \to s$, i.e. a loop. Thus, the period of s is 1; the irreducible chain is aperiodic. □

4.4. Accuracy of the Model

There is only one point in our model that may introduce inaccuracy in addition to the one inherent to the (history) stack histogram: the handling of accesses on elements with cutoff age. We must introduce a cutoff age to obtain a finite model and due to efficiency reasons (see Section 5). Thus, we have to make assumptions about the actual ages of elements having the abstract cutoff age. As specified in the second case in Section 4.2, we assume that the probability that a cutoff age element has age i decreases exponentially. We assign each cutoff age element the same hit probability p_{hit} because we have no information about their order; every element could be the youngest.

5. Evaluation

In this section we present results on the accuracy of the predicted miss ratios and the runtime of our algorithm. We also establish a connection to so-called replacement probability functions, which were introduced in [12], and show how to compute them.

		LRU	PLRU			FIFO			MRU			RAND		
Benchmark	Input		Sim	0/20	1/20	Sim	0/15	1/10	Sim	0/19	1/14	Sim	0/11	1/8
164.gzip	graphic	3.90	3.91	0.08	0.07	5.10	-0.67	-0.70	16.64	3.68	4.83	4.76	0.32	0.33
164.gzip	log	3.40	3.27	0.17	0.17	4.10	-0.44	-0.46	12.01	4.38	3.43	3.93	0.11	0.12
164.gzip	program	0.90	0.90	0.04	0.04	1.19	-0.15	-0.16	4.30	5.25	2.81	1.12	0.08	0.07
164.gzip	random	4.02	4.03	0.08	0.07	5.15	-0.65	-0.69	17.01	3.15	4.61	4.80	0.29	0.30
164.gzip	source	1.65	1.62	0.11	0.11	2.15	-0.28	-0.29	6.71	5.65	3.75	2.02	0.08	0.09
175.vpr	place	18.59	19.13	0.03	0.02	22.22	-1.72	-1.62	30.77	-0.18	-0.14	21.25	0.44	1.40
175.vpr	route	28.89	29.24	0.06	-0.01	30.47	-0.22	-0.37	39.87	0.85	-0.23	30.74	0.83	0.88
176.gcc	166	26.43	26.42	0.94	0.54	26.71	2.77	1.68	34.87	-3.00	-1.17	22.47	8.38	6.52
176.gcc	200	8.57	8.48	0.84	0.54	8.96	1.39	0.83	16.78	6.16	2.90	7.57	3.73	3.20
176.gcc	expr	2.83	2.82	0.81	0.53	3.09	1.56	0.99	14.66	6.36	3.09	2.99	2.73	2.16
176.gcc	integrate	7.20	6.43	2.86	2.42	8.32	2.32	1.79	26.62	-0.86	-1.65	5.41	6.69	8.39
176.gcc	scilab	8.98	8.93	0.21	0.01	9.25	0.94	0.59	15.68	6.30	3.37	6.84	3.73	3.70
177.mesa	ref	5.18	5.19	-0.00	-0.01	5.34	-0.09	-0.12	7.39	6.03	2.79	5.24	0.23	0.19
179.art	ref.1	79.32	79.32	-0.04	-0.03	79.32	-0.08	-0.05	57.30	-1.42	-3.43	56.81	8.11	15.00
179.art	ref.2	79.10	79.10	-0.04	-0.03	79.10	-0.08	-0.05	57.26	-1.54	-3.49	56.73	8.03	14.89
181.mcf	ref	67.43	67.47	-0.24	-0.36	67.34	-0.27	-0.63	70.27	-4.21	-4.70	62.34	4.12	3.29
183.equake	ref	76.44	76.44	-0.00	-0.00	76.45	-0.01	-0.01	77.24	0.82	0.30	76.42	0.02	-0.02
188.ammp	ref	43.76	43.80	0.01	-0.04	44.01	0.09	-0.13	47.24	5.17	1.91	43.57	1.12	0.62
197.parser	ref	19.39	19.47	0.38	0.24	20.20	0.68	0.32	28.82	4.20	1.89	20.75	1.34	1.11
253.perlbmk	diffmail	0.50	0.50	0.02	0.02	0.53	0.08	0.05	2.00	8.22	3.38	0.52	0.23	0.16
253.perlbmk	makerand	0.02	0.03	-0.01	-0.01	0.03	-0.01	-0.01	0.14	1.52	0.00	0.03	0.02	-0.00
253.perlbmk	perfect	0.00	0.03	0.02	0.05	0.00	0.04	0.06	2.85	3.10	4.06	0.00	0.20	0.15
253.perlbmk	splitmail_535	2.40	2.43	0.10	0.07	2.91	-0.15	-0.23	5.48	7.95	1.52	2.77	0.27	0.18
253.perlbmk	splitmail_704	2.88	2.92	0.10	0.07	3.48	-0.14	-0.24	6.62	8.31	2.23	3.25	0.40	0.32
253.perlbmk	splitmail_850	2.30	2.29	0.06	0.04	2.62	-0.10	-0.18	4.60	7.58	0.97	2.44	0.24	0.17
253.perlbmk	splitmail_957	2.14	2.16	0.12	0.09	2.65	-0.16	-0.24	4.98	7.81	1.31	2.50	0.24	0.17
255.vortex	lendian1	3.54	3.69	0.17	0.13	5.06	-0.54	-0.65	12.26	7.07	1.84	4.59	0.60	0.51
255.vortex	lendian2	3.10	3.18	0.13	0.08	4.11	-0.30	-0.43	8.84	7.10	1.13	3.73	0.53	0.44
255.vortex	lendian3	3.61	3.75	0.16	0.13	5.15	-0.59	-0.69	12.57	6.99	1.91	4.66	0.57	0.49
256.bzip2	graphic	35.10	35.41	0.07	-0.01	36.96	-0.66	-0.84	46.36	-1.36	-2.89	37.11	0.47	0.40
256.bzip2	program	34.50	34.70	0.08	0.02	35.93	-0.36	-0.50	42.93	1.29	0.60	36.07	0.58	0.57
256.bzip2	source	35.57	35.77	0.05	-0.03	36.95	-0.34	-0.51	42.06	2.70	1.68	37.18	0.58	0.63
300.twolf	ref	22.96	23.33	0.10	0.06	26.15	-1.44	-1.36	33.54	0.31	0.42	25.29	0.66	1.57
Averages		19.23	19.28	0.25	0.18	20.03	0.59	0.53	24.44	4.26	2.26	18.06	1.70	2.06

Table 1. Accuracy of miss-ratio estimates for a 512kB cache (8-way, 2048 sets, 32 byte line size).

5.1. Precision

We compared the estimates obtained with our model with ones obtained by simulation using SimpleScalar [2]. The SimpleScalar tool set can be used to build modeling applications for program performance analysis including detailed micro-architectural modeling. We exchanged its built-in cache module with one that is capable of handling our policy tables and added code to record history stack histograms. We took C programs from the SPEC CPU2000 benchmark suite and recorded history stack histograms of shared 2nd-level caches, which usually exhibit higher miss rates and are more sensitive to the replacement policy. We used these histograms as inputs to our model and predicted miss ratios for several cache configurations and replacement policies. For the policies we picked LRU, PLRU, FIFO, MRU, and RAND introduced in Figure 2.

Table 1 shows the main results for an 8-way associative cache of size 512kB.[1] We decided to give the bare numbers in this form since the contained information cannot be compactly represented graphically. Only giving average values would conceal the (in)accuracy of individual estimations. Each table compares simulation with estimation results for different policies for all SPEC-reference inputs. The "Sim" columns contain miss ratios obtained by simulation. The "l/c" columns contain absolute errors of the estimated miss ratios obtained by using our model with history length l and cutoff-age c (e.g. 1/20). Finally, the last row gives averages for all columns. Note, that for the l/c columns the average of *absolute* values ($\frac{1}{n}\sum|\text{Est}_i - \text{Sim}_i|$) is given, such that under- and overestimations do not cancel each other out.

The l/c settings were chosen maximally such that the models still fit in 2GB of main memory, but not larger than $c = 20$. There are two notable exceptions: Since LRU miss ratios can be precisely predicted there is only one column for LRU. For FIFO, there also is a provable upper bound on the age of cached elements: For associativity k all cached elements must have an age $\leq 2k - 2$. Hence, cutoff age $2k - 1$ is sufficient to completely disambiguate the ages of cached elements. In this case, the cutoff age does not occur in the ages tuples of the model and increasing the cutoff age further will not change the model. For all other investigated policies the age of a cached element is unbounded.

The miss ratios vary a lot: Over all benchmarks they range from almost 0% up to almost 80%. The differences between policies for each particular benchmark are at most 23% absolute difference (Table 1, `179.art`). In this particular case MRU and RAND perform much better than LRU-like policies since a good share of accesses have stack distance slightly larger than the associativity ($k = 8$) of the cache (see Figure 1(d), and Figure 6). Another case where RAND performs better than LRU is `176.gcc`.

Given the fact that stack histograms already exhibit an inherent amount of inaccuracy, as discussed in Section 2.1, the prediction results are good: The average absolute[2] prediction error is between 0.18% and 2.92% for 8-way models

[1] Due to space limitations we had to remove result tables for other cache configurations. These can be found at `rw4.cs.uni-sb.de/~grund`.

[2] We give *absolute* errors because for small miss ratios even one small prediction error would lead to a seemingly high average *relative* error.

l/c		PLRU	FIFO	MRU	RAND
0/8	#states	2391	265545	2737	453118
	∅runtime [s]	0.05	7.15	0.06	43.52
1/8	#states	17798	2195376	15626	2687856
	∅runtime [s]	0.43	90.14	0.39	234.58

(a) Influence of history length with fixed cutoff age l.

	PLRU	FIFO	MRU	RAND
0/max c	0/20	0/15	0/19	0/11
#states	450627	2027025	6210719	7064350
∅runtime [s]	22	86	400	1114
1/max c	1/20	1/10	1/14	1/8
#states	4166048	8216745	4129362	2687856
∅runtime [s]	386	440	263	234

(b) Maximal sensible or feasible (for 2GB RAM) parameter settings.

Figure 5. Number of Markov chain states and corresponding average run-times (in seconds) of our algorithm for 8-way associative caches dependent on history length l and cutoff age c.

with history. The exceptionally good results for PLRU (in contrast to other policies) might be explained by the fact that LRU can be predicted precisely and PLRU behaves similarly to LRU. We conjecture that the comparatively high estimation errors for RAND can be attributed to the low cutoff age. The improvement observed when going from cutoff age 8 to 11 without history suggests further improvements with higher cutoff ages.

We performed estimations with all possible cutoff ages to assess its influence on the precision. Significant improvements in the prediction error are visible up to a cutoff age of $\frac{3}{2}k$ to $2k$, depending on the policy. It is very unlikely that elements of such high ages are present in the cache.

In many cases the size of main memory limits the model parameters. There is a trade-off between history length and cutoff age. Except for the RAND case just mentioned, increasing (enabling) the history length yields higher precision than maximizing the cutoff age without history. Enabling history while keeping the cutoff age constant always improves precision. See below for details on model sizes.

We also ran simulations (and estimations) for 4-way associative, 2048-set caches. In general the miss ratios are higher and the accuracy of the model is better. The advantage of MRU and RAND in the special cases discussed above still exists, but is smaller. The results are better because the differences between policies vanish with lower associativity. In addition, higher cutoff ages can be chosen because the models are significantly smaller for a given cutoff age, due to the smaller associativity.

5.2. Memory Consumption and Runtime

The number of states in the Markov chain depends on the given policy table, history length, and cutoff age. The transition probabilities depend on the policy table and stack histogram. Table 5 gives the number of states in the Markov chain and average run-times (Pentium 4, 2.66GHz) of our method for several configurations. In a model for LRU there is only one possible ages tuple. The number of states is solely determined by history length. Depending on the policy the number of states varies greatly, see Table 5(a). In case of PLRU relatively few states arise. This nicely reflects the fact that PLRU approximates LRU. Introducing history greatly increases the number of states for all policies. Increasing the history length by one maximally increases the state space by a factor of c. As such an increase is in fact observed in our examples, history lengths greater than one are currently not conceivable.

In Table 5(b), we illustrate the maximal parameter settings that are possible with 2 GB of main memory. RAND strikes out by particularly high memory consumption. As it does not follow a particular strategy to retain elements of low age, many cache-set states are possible.

The runtime of our method (again using the same SPEC benchmarks) strongly depends on the policy employed. The time to build the Markov model dominates the total runtime of the estimation process. Therefore it is strongly correlated to the number of states. In addition to the number of states and transitions, the time spent to compute the steady state depends on the transitions probabilities and the ordering of states in the transition matrix. This explains why e.g. FIFO-1/10 with 8 million states needs on average less time than RAND-0/11 with only 7 million states. In contrast to an estimation, which takes seconds to some minutes, an average simulation run with SimpleScalar takes about 18 hours.

Summarizing, one can say that associativity 4 is handled without any problems. For associativity 8 and without history LRU, PLRU, and FIFO can be estimated using the maximal sensible cutoff age. When enabling history the cutoff age must be reduced for FIFO and MRU but the resulting estimates are still better than with maximal cutoff age but no history. RAND with its unstructured policy table only allows low parameter settings but the estimates are still good with some exceptions.

5.3. Replacement Probability Functions

Guo and Solihin [12] proposed replacement probability functions to model the behavior of replacement policies.

Definition 6. *A replacement probability function (RPF) is a function $r : \mathbb{N} \to [0, 1]$. It gives the conditional probabilities $r(n)$ that an element of age n is replaced given that a cache miss happens.*

They propose a performance prediction model [12] that uses these RPFs together with a locality-characterization called circular sequence profiles [9]. There are two problems with this approach: Firstly, to obtain miss-ratio estimates for a new policy one has to provide an RPF to the model. Without theoretical reasoning or simulations it is difficult to say how a (new) policy will behave, i.e. the shape of the corresponding RPF is unclear. More importantly, RPFs do not only depend on the replacement policy; RPFs *also depend* on the program being executed and its input. Hence, an RPF does not solely characterize a policy, but a combination of policy and program. Thus, strictly speaking, the input required by their model depends on replacement behavior, which is (part of) the output of the model. Due to this cyclic dependency, the RPFs used as inputs can only be heuristic estimations. Figure 6 shows some RPFs and illustrates their dependency on the executed program.

In the following we show how accurate RPFs can be computed from intermediate results of our model. Assume one has already computed the steady-state probabilities p and miss probabilities m of our model as described above. Let p_s and m_s be the corresponding probabilities for a given state s. First, one has to consider for all states the age of the element at position 0, the replacement position. Let $S|_n := \{s \in S \mid s = ([n, a_1, \ldots, a_{k-1}], h)\}$ be the subset of states that have an element of age n at position 0. Then,

$$r'(n) := \sum_{s \in S|_n} p_s \cdot m_s$$

is the probability that a miss happens and replaces an element of age n. Finally, the RPF $r(n)$ is obtained by normalizing $r'(n)$ with $C := \sum_i r'(i)$:

$$r(n) := \frac{1}{C} \cdot r'(n)$$

6. Related Work

In their fundamental article, Mattson et al. [14] in 1970 introduced stack histograms and stack algorithms. Stack algorithms are replacement algorithms that satisfy their inclusion property: Larger buffers for all points in time always subsume the contents of smaller buffers. Given a stack algorithm and an address trace their method is able to derive miss ratios for different sizes of buffers. Although the inclusion property does not hold for e.g. FIFO and PLRU we can still make use of the very basic results, i.e. representing locality information as stack histograms.

Hill and Smith [13] investigate the effect of associativity on the miss ratio. They introduce forest simulation and all-associativity simulation. The first can be used to determine miss ratios for alternative direct-mapped caches. The latter is restricted to LRU replacement but is able to predict miss ratios for different numbers of sets and associativities with low overhead. As mentioned in Section 2 one can use this to efficiently obtain stack histograms for different numbers of sets. These stack histograms can then be used by our method to estimate miss-rates for other policies as well.

Berg et al. [6, 5] compute miss ratios for fully associative caches with truly random replacement. They use a slightly different definition of reuse distance in which duplicate accesses contribute to the reuse distance. Stack histograms based on this definition can be efficiently determined by hardware counters and hardware watch points. Clearly, this work focuses on efficient data-locality profiling.

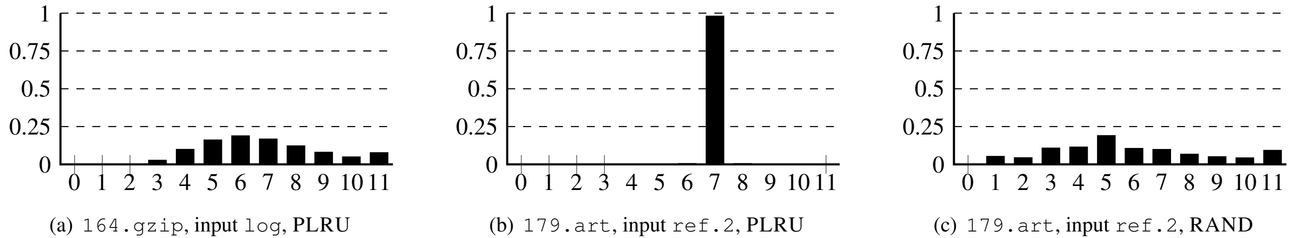
Cascaval and Padua [8] compute stack histograms during compile-time. They use the obtained information to guide optimizations that improve data locality. This work is focused on loops with array references and assumes LRU as replacement policy. Because a stack histogram is available one could extend their work with our model to handle other policies as well.

Directly related to the performance of different policies are the last two papers we want to discuss. The first by Al-Zoubi et al. [1] is a thorough measurement based investigation. The paper considers the behavior of LRU, FIFO, PLRU, pseudo round robin (called random), and another kind of LRU approximation on SPEC benchmarks.

The work closest to ours is by Guo and Solihin [12]: they also propose a model to predict replacement policy performance. Unlike us, they use circular-sequence profiles instead of (history) stack histograms and replacement probability functions (RPF) instead of policy tables. These differences lead to fundamentally different models. More importantly, besides LRU, they only consider virtual policies specified by RPFs. In general (LRU is an exception), it is impossible to define a policy by an RPF that is independent of the workload. For details confer Section 5.3. Instead we are the first to consider widely used policies and characterize them in a form that is independent of the workload. A direct comparison of attained precision is impossible because the only common policy is LRU and they only report their data in graphical form or as average values.

7. Summary and Future Work

In this paper, we demonstrated how to combine stack histograms and policy tables to obtain a stochastic model of cache behavior. Its main application is the estimation of miss ratios. The average absolute prediction error is between 0.18% and 2.92% for widely used policies on SPEC workloads. However, the model can also be used to determine other properties about the cache behavior of a program: For instance, replacement probability functions, distributions over ages of cached elements, and average values like the mean residence time of a memory block.



(a) `164.gzip`, input `log`, PLRU (b) `179.art`, input `ref.2`, PLRU (c) `179.art`, input `ref.2`, RAND

Figure 6. Replacement probability functions for: a),b) different programs with same policy; b),c) same program with different policies.

We discussed the inaccuracy inherent to the input, the stack histograms, and additional inaccuracy introduced by the cutoff age of our model. To reduce inaccuracy in the inputs, we introduced history stack histograms that add negligible overhead to their computation and storage. Our evaluation shows that history significantly improves the precision of estimates. The inaccuracy introduced by the cutoff age vanishes if it is chosen to be greater than two times the associativity. We assume that longer history lengths (to a limited extend) would result in better estimates. To increase history length, one would have to use a stronger abstraction for other parts of the model to make up for the incurred additional memory demand.

Another extensions is to consider other replacement policies. These can be included in our model by defining a policy table for them or by changing the cache-set state representation in the Markov chain states. Finally, the policy tables span a large space of policies, in which one could search for policies satisfying certain properties (besides performance).

References

[1] H. Al-Zoubi, A. Milenkovic, and M. Milenkovic. Performance evaluation of cache replacement policies for the SPEC CPU2000 benchmark suite. In *ACM-SE 42: Proceedings of the 42nd Annual Southeast Regional Conference*, pages 267–272, New York, NY, USA, 2004.

[2] T. Austin. Simple Scalar. http://www.simplescalar.com/.

[3] R. Barrett, M. Berry, T. F. Chan, J. Demmel, J. M. Donato, J. Dongarra, V. Eijkhout, R. Pozo, C. Romine, and H. Van der Vorst. *Templates for the Solution of Linear Systems: Building Blocks for Iterative Methods*. Society for Industrial and Applied Mathematics, Philadelphia, PA, 1994.

[4] C. Berg. PLRU cache domino effects. In *6th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, 2006.

[5] E. Berg and E. Hagersten. Fast data-locality profiling of native execution. In *Proceedings of SIGMETRICS 2005*, pages 169–180, New York, NY, USA, 2005.

[6] E. Berg, H. Zeffer, and E. Hagersten. A statistical multiprocessor cache model. In *Proceedings of ISPASS 2006*, Austin, Texas, USA, Mar. 2006.

[7] P. Bose and T. M. Conte. Performance analysis and its impact on design. *Computer*, 31(5):41–49, 1998.

[8] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of ICS 2003*, pages 150–159, New York, NY, USA, 2003.

[9] D. Chandra, F. Guo, S. Kim, and Y. Solihin. Predicting inter-thread cache contention on a chip multi-processor architecture. In *Proceedings of HPCA 2005*, pages 340–351, Washington, DC, USA, 2005.

[10] C. Ding and Y. Zhong. Predicting whole-program locality through reuse distance analysis. In *Proceedings of PLDI 2003*, pages 245–257, New York, NY, USA, 2003.

[11] C. Fang, S. Carr, S. Önder, and Z. Wang. Reuse-distance-based miss-rate prediction on a per instruction basis. In *Proceedings of MSP 2004*, New York, NY, USA, 2004.

[12] F. Guo and Y. Solihin. An analytical model for cache replacement policy performance. *SIGMETRICS Perform. Eval. Rev.*, 34(1):228–239, 2006.

[13] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Trans. Comput.*, 38(12):1612–1630, 1989.

[14] R. L. Mattson, J. Gecsei, D. R. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM Systems Journal*, 9(2):78–117, 1970.

[15] P. Nagpurkar, C. Krintz, M. Hind, P. F. Sweeney, and V. T. Rajan. Online phase detection algorithms. In *Proceedings of CGO 2006*, pages 111–123, Washington, DC, USA, 2006.

[16] J. J. Pieper, A. Mellan, J. M. Paul, D. E. Thomas, and F. Karim. High level cache simulation for heterogeneous multiprocessors. In *Proceedings of DAC 2004*, pages 287–292, New York, NY, USA, 2004.

[17] J. Reineke, D. Grund, C. Berg, and R. Wilhelm. Timing predictability of cache replacement policies. *Real-Time Systems*, 37(2):99–122, November 2007.

[18] X. Shen, Y. Zhong, and C. Ding. Predicting locality phases for dynamic memory optimization. *J. Parallel Distrib. Comput.*, 67(7):783–796, 2007.

[19] A. J. Smith. A comparative study of set associative memory mapping algorithms and their use for cache and main memory. *IEEE Trans. Software Eng.*, 4(2):121–130, 1978.