On the Theory and Potential of LRU-MRU Collaborative Cache Management

Xiaoming Gu   Chen Ding

Department of Computer Science
University of Rochester
Rochester, New York, USA
{xiaoming, cding}@cs.rochester.edu

Abstract

The goal of cache management is to maximize data reuse. Collaborative caching provides an interface for software to communicate access information to hardware. In theory, it can obtain optimal cache performance.

In this paper, we study a collaborative caching system that allows a program to choose different caching methods for its data. As an interface, it may be used in arbitrary ways, sometimes optimal but probably suboptimal most times and even counter productive. We develop a theoretical foundation for collaborative caches to show the inclusion principle and the existence of a distance metric we call LRU-MRU stack distance. The new stack distance is important for program analysis and transformation to target a hierarchical collaborative cache system rather than a single cache configuration. We use 10 benchmark programs to show that optimal caching may reduce the average miss ratio by 24%, and a simple feedback-driven compilation technique can utilize collaborative cache to realize 50% of the optimal improvement.

Categories and Subject Descriptors D.3.4 [*PROGRAMMING LANGUAGES*]: Processors - Compilers, Memory management

General Terms Algorithms, Measurement, Performance

Keywords collaborative caching, bipartite cache, cache replacement algorithm, LRU, MRU, OPT

1. Introduction

Cache management is increasingly important on multicore systems since the available cache space is shared by an increasing number of cores. Optimal caching is generally impossible at the system or hardware level for lack of program information. At the program level, optimal caching requires solving NP-hard problems and is not yet practical [14, 20, 25].

A number of hardware systems have been built or proposed to provide an interface for software to influence cache management. Examples include cache hints on Intel Itanium [9], bypassing access on IBM Power series [27], and evict-me bit [31]. Our earlier work showed a theoretical result that two extensions of LRU cache may be managed optimally by a program [17]. Wang et al. called a combined software-hardware solution *collaborative caching* [31].

In this paper, we study the formal properties of collaborative cache management. We define a model called *bipartite cache*. It supports two types of accesses: the normal LRU access and the special MRU access. Data loaded by an MRU access is managed by MRU replacement. At a miss, it selects the most recently used data for eviction. It is equivalent to tagging the loaded data with an evict-me flag [31], setting the MRU data for eviction before any LRU data. With bipartite cache, a program influences the cache management by selecting which data to be accessed by which type.

In comparison, conventional cache uses a variant of the LRU strategy. More significantly, a conventional cache uses a single interface for all data access. The use of software control makes hybrid management inevitable. The LRU-MRU combination in one cache warrants a re-examination of the fundamental properties of caching.

A foremost property of memory (and storage) hierarchy is what Mattson et al. termed the *inclusion principle*, which says that the content of a smaller cache is always contained in larger caches [23]. The inclusion principle has important benefits. In theory, the miss ratio is a monotone function of cache size. There is no Belady anomaly [7]. In practice, the miss ratio of caches of all sizes can be evaluated using one-pass simulation over a program trace.

More importantly for software, a machine-independent metric called stack distance can be defined for each access [23]. Software techniques are developed to minimize the stack distance and improve performance for a cache hierarchy rather than targeting a particular cache level. A type of stack distance called reuse distance has been used to improve memory management including garbage collection techniques [34, 36].

The inclusion principle is intuitive for LRU cache. Data is ranked by the last access time. The most recent data enters from the top of the cache stack and gradually steps down as it ages over time. Bipartite cache, however, stores data in two parts and ranks data in opposite ways. The MRU data is placed at the bottom of the cache. The placement depends on cache size. As Mattson et al. have cautioned, it goes against the inclusion principle to base a caching decision on cache capacity [23].

In this paper, we analyze the general LRU-MRU bipartite cache. Interestingly, the inclusion property still holds. We give a proof first and then an efficient one-pass simulation algorithm to compute the miss ratio for fully associative caches of all sizes. The algorithm defines and measures the *LRU-MRU stack distance* for each access. An access is a hit in bipartite cache if and only if its LRU-MRU stack distance is no greater than the cache size.

To demonstrate the potential benefit of bipartite cache, we describe a simple method called *PACMAN* for Program-Assisted

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

ISMM'11, June 4–5, 2011, San Jose, California, USA.

Copyright © 2011 ACM 978-1-4503-0263-0/11/06...$10.00

Cache MANagement. PACMAN shows how much a program can reduce the miss ratio under a number of simplifying assumptions. It shows the potential of collaborative cache management but it is not yet a practical solution.

In practice, caches are set associative rather than fully associative. For theoretical analysis, fully associative cache is more interesting (and difficult) since the associativity changes with the cache size. Its properties and results have practical significance. First, the inclusion property holds for each set of set-associative cache and for most real cache hierarchies. Second, modern cache has high associativity, i.e. 8-way and up, which means similar performance as fully associative cache [18]. The empirical results in the paper show the general effect of bipartite cache of all sizes, regardless of the specific implementation. Finally, important for software research, the LRU-MRU stack distance provides a machine-independent target for program analysis and transformation, as we explain in Section 5.

2. Background

LRU The data in cache is sorted by the last use time. If x is a miss and the cache is full, the datum in the LRU (least recently used) position is evicted. LRU can be costly when the set associativity is high. Pseudo-LRU is the one usually used in practice [29]. The performance of fully associative LRU cache can be measured in one pass for all cache sizes using reuse distance analysis in near linear time [37].

OPT The *optimal replacement algorithm (OPT)* was invented by Belady [6]. At a replacement, the victim is the datum that will be reused in the farthest future. Mattson et al. showed its inclusion property and gave a two-pass algorithm to measure the OPT stack distance [23]. Sugumar and Abraham invented an efficient one-pass algorithm [30]. OPT is not practical purely in hardware as it would require infinite ahead. However, it has a vital theoretical value since it shows the limit of caching.

Stack algorithm and stack distance As defined by Mattson et al., if the content of a smaller cache is always a subset of the content of a larger cache, the cache management algorithm obeys the *inclusion property* and is considered a *stack algorithm* [23]. In stack algorithms each access has a *stack distance*, the minimum cache size, to make the access become a hit. The miss rate is a monotone (non-increasing) function of cache size, and there is no Belady's anomaly [7]. The paper proved the inclusion property for LRU, LFU (least frequently used), OPT, and a form of random replacement. The inclusion property is beneficial in evaluation, when we can evaluate all cache sizes in one simulation, and in analysis, when we can target the stack distance and all cache sizes instead of a single cache size.

Two characteristics of the LRU-OPT gap As an example, we show the difference between LRU and OPT cache replacement algorithms using a workload of Jacobi Successive Over-relaxation (SOR) from SciMark 2.0 [2]. We use currently the fastest one-pass analysis methods for LRU [37] and OPT [30]. The LRU and OPT miss rate curves of an execution of SOR are shown in Figure 1 for cache sizes ranging between 1KB and 8MB (twice the size of the program data). The cache line size is 8 bytes.

Figure 1 shows two interesting aspects of optimal caching compared with LRU.

- *Non-uniform improvement.* OPT is not uniformly better than LRU. The improvement varies greatly between cache sizes.
- *Gradual miss-ratio change.* The miss ratio of OPT decreases gradually as the size of cache increases.

We observe that the curves of OPT and LRU diverge first, converge at size 16KB and then diverge again before both dropping to near zero at 4MB (with only cold-start misses). The difference depends on the cache size. In 16KB or 32KB cache, there is little or no improvement. In 8KB and 2MB cache, the improvement is more than 60% and 90% respectively. It is important to evaluate across all cache sizes.

The OPT miss ratio changes gradually, while the LRU miss ratio either stays the same or drops sharply. The sharp drops mark the size of working sets—each steep descent happens when the cache is large enough to hold the next working set. SOR has mainly two working sets: one at 8KB and one at 2MB. The smooth curvature of OPT shows that it caches a partial working set if the whole set is too large.

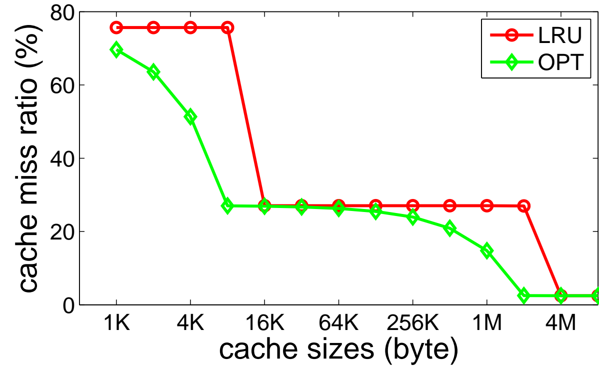


Figure 1. The gap between LRU and OPT in SOR

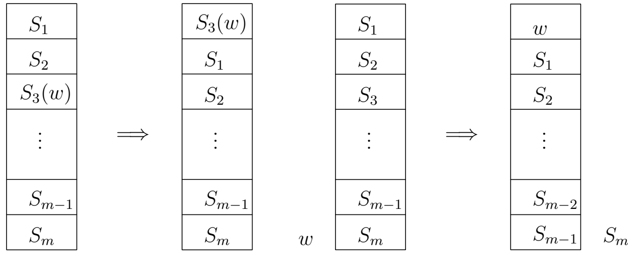
Collaborative caching In collaborative caching, a program designates some of its references to make MRU accesses. Figure 2 shows the kernel SOR whose miss rates are just shown in Figure 1. It is typical of stencil algorithms. Consider the data access in the loop body. Array `G` is traversed in each iteration of the outermost loop. If `M*N` is larger than cache size, array `G` cannot fit entirely in the cache. The streaming access of `G` would lose all data reuse because LRU evicts the least recently used datum, which is actually the datum that will be reused in the nearest future. OPT, however, would evict the most recently used datum. To obtain the same effect, we can tag the last access to each datum as an MRU access. A bipartite cache of size C would keep the first C bytes of `G` in cache and reuse them across loop iterations.

```
Require: G is a 2-dimensional double array with the size M*N
 1: for p = 1; p < NUM_STEPS; p++ do
 2:     for i = 1; i < M-1; i++ do
 3:         Gi = G[i];
 4:         Gim1 = G[i-1];
 5:         Gip1 = G[i+1];
 6:         for j = 1; j < N-1; j++ do
 7:             Gi[j] = 0.3125*(Gim1[j]+Gip1[j]+Gi[j-1]+Gi[j+1])-0.25*Gi[j];
 8:         end for
 9:     end for
10: end for
```

Figure 2. The SOR kernel computation

3. Properties of Collaborative Cache

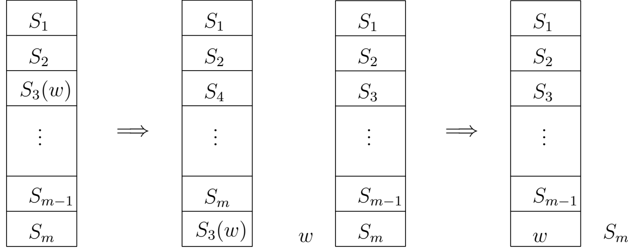
We first define the LRU and MRU memory accesses and then prove that bipartite cache has the inclusion property and can be evaluated efficiently using one-pass simulation.



(a) A LRU hit: w, assuming at entry S_3, is moved to the top of the stack

(b) A LRU miss: w is placed at the top of the stack, evicting S_m

Figure 3. A LRU memory access



(a) An MRU hit: w, assuming at entry S_3, is moved to the bottom of the stack

(b) An MRU miss: w is placed at the bottom of the stack, evicting S_m

Figure 4. An MRU memory access

3.1 Bipartite LRU-MRU Cache

The collaborative cache provides two instructions for accessing memory: the normal LRU access and the special MRU access. LRU is a standard concept defined in textbooks. For comparison with MRU, we show a diagram in Figure 3. LRU cache can be thought of as organized in a stack. The newly accessed data is at the top—the MRU position—and the rest of LRU data is ordered top-down based on the recency of access. In an actual implementation, the order of recency is maintained or approximated efficiently without moving cache entries.

In comparison, Figure 4 shows the handling of an MRU access. If it is a miss, the new data is replaced at the LRU position at the bottom of the stack. If it is a hit, the accessed data is moved to the bottom of the stack. Multiple MRU elements may gather at the bottom after a series of MRU access hits.

The LRU-MRU interface can be used to obtain optimal cache performance [17], yet it is simple to implement. The implementation of the MRU instruction is not much harder than a normal LRU instruction. In a real hardware design, the cycles required to execute an MRU access should be similar to the cost of a normal LRU cache access.

Bipartite cache differs from conventional cache in three ways:

- *Bipartite content.* The cache stack is divided into two parts: the upper part for LRU data and the lower part for MRU data. Either part may be missing, and the cache is entirely LRU or MRU.
- *Capacity dependent placement.* The MRU data is placed at the bottom. The location depends on the size of the cache.
- *Hybrid priority.* The LRU part is prioritized by the LRU order, that is, the last accessed is last replaced. The MRU part is by the MRU order, that is, the last accessed is first replaced.

In comparison, conventional, non-collaborative cache manages data using a single priority order, e.g. LRU by the last access time and OPT by the next access time. The placement depends on the priority order and not on cache size. The single priority naturally gives rise to the inclusion property and its practical benefits. To understand collaborative caching, we must understand its bipartite nature.

3.2 The Inclusion Property

If the collaborative cache is used optimally, the performance is the same as OPT [17]. In general, however, the cache may not be used optimally. The selection of MRU accesses may be arbitrary. The following proof is for all uses of bipartite cache, including the extreme cases (when all accesses are normal, i.e. LRU caching, and when all accesses are special, i.e. MRU caching), the optimal use, and everything in between. In a sense, the proof subsumes the individual conclusions for LRU, MRU, and OPT [23].

We prove that for any sequence of LRU and MRU accesses, the bipartite cache obeys the inclusion principle.

LEMMA 1. *If the bottom element in the bipartite cache stack is last accessed by a normal LRU access, then all elements in cache are last accessed by normal LRU accesses.*

The Lemma 1 follows from the fact that MRU data are placed at the bottom of the stack and only replaced by LRU data (never pushed up except by other MRU data). There is a formal proof of Lemma 1 in our workshop paper [17]. Next we prove the inclusion property.

THEOREM 1. *A trace P is being executed on two bipartite caches of sizes $|C_1|$ and $|C_2|$ ($|C_1| < |C_2|$). At every access, the content of cache C_1 is always a subset of the content of cache C_2.*

Proof Let the access trace be $P = (x_1, x_2, ..., x_n)$. Let $C_1(x_t)$ and $C_2(x_t)$ be the set of elements in cache C_1 and C_2 after access x_t. The initial cache contents are $C_1(0) = C_2(0) = \emptyset$. The inclusion property holds. We now prove the theorem by induction on t.

Assume $C_1(x_t) \subseteq C_2(x_t)$ $(1 \leq t \leq n-1)$. It is easy to see that if x_{t+1} is a hit in C_2 ($x_{t+1} \in C_2(x_t)$), the inclusion property holds. We now consider the case that x_{t+1} is a miss in C_2. Since C_1 is included in C_2, x_{t+1} is also a miss in C_1.

Let the evicted elements be last accessed at x_p in C_1 and x_q in C_2. After the cache miss, we have $C_1(x_{t+1}) = C_1(x_t) - x_p + x_{t+1}$ and $C_2(x_{t+1}) = C_2(x_t) - x_q + x_{t+1}$. Since $C_1(x_t) \subseteq C_2(x_t)$, the only possibility for $C_1(x_{t+1}) \nsubseteq C_2(x_{t+1})$ is that C_2 evicts x_q, and C_1 has x_q but does not evict it, so $x_q \in C_1(x_{t+1})$ but $x_q \notin C_2(x_{t+1})$.

First we assume x_p exists (a cache miss does not mean a cache eviction—see the next case). The eviction in C_1 happens at the LRU position regardless whether x_p is a LRU or MRU access. x_p is at the bottom in C_1 before access x_{t+1}. At the same time, x_q is at the bottom in C_2. To violate the inclusion property, we must have $x_q \in C_1(x_t)$ in a position over x_p. From the inductive assumption, $x_p \in C_2(x_t)$ and it is in a position over x_q. Therefore, both C_1 and C_2 contain x_p and x_q but in an opposite order.

The two accesses, x_p and x_q, may be LRU or MRU accesses. There are four cases:

I x_p and x_q are both LRU accesses. Because x_q is at a higher position than x_p in C_1, we have $p < q$. Similar reasoning from C_2 requires $q < p$, which makes this case impossible.

II x_p is normal, and x_q is a MRU access. Using Lemma 1 on C_1, we see that this case is impossible—x_q has to be normal because it resides over a normal access x_p in C_1.

III x_p is a MRU access, and x_q is normal. Using Lemma 1 on C_2, we see that this case is impossible—x_p has to be normal because it resides over a normal access x_q in C_2.

IV x_p and x_q are both MRU accesses. Because x_q is at a higher position than x_p in C_1, we have $p > q$. Similar reasoning from C_2 requires $q > p$, which makes the last case impossible.

There is no eviction in C_1 if the bottom cache line is unoccupied when x_{t+1} is accessed. x_q is at the bottom of C_2. Regardless of whether x_q is LRU or MRU, C_2 is filled. Since $|C_2| > |C_1|$, there must have been enough data access to fill C_1, making it impossible for its bottom spot to remain unoccupied. Hence, by induction, the inclusion property holds for every access in the trace. ∎

The inclusion property holds for any access trace with mixed LRU and MRU accesses, regardless how these two types of accesses are interleaved.

3.3 The LRU-MRU Stack Distance

The inclusion property implies the existence of the LRU-MRU stack distance. An access has a distance k if it is a cache hit in caches of sizes k and up and a miss in caches of size $k - 1$ and down. Given a program trace with mixed LRU-MRU accesses, Algorithm 1 computes the stack distance for each access. Effectively the algorithm simulates LRU-MRU caches of *all* sizes—top C elements in the priority list are always the content of a cache with size C. We call the algorithm *bi-sim* in short for bipartite cache simulation.

Algorithm 1 Bi-sim: computing the stack distance of bipartite cache

Require: x is accessed at time t with flag $f = \{LRU, MRU\}$. The cache is organized as a priority list, with data d_i and priority p_i, $i = 1, \ldots, M$. No two priorities are the same, i.e. $\forall i$ and j, $p_i \neq p_j$ if $i \neq j$. The list may not have been sorted.

Ensure: It returns the LRU-MRU stack distance and updates the priority p_x of x (first adding it to the priority list if it was not included). The priority p_x is unique.

```
1:  if f = LRU then
2:      p_x = t
3:  else
4:      p_x = −t
5:  end if
6:  /* process x */
7:  if x ∉ {d_i : i = 1, ..., M} then
8:      /* x is a miss */
9:      for i = 1; i < M; i++ do
10:         if p_i < p_{i+1} then
11:             swap d_i and d_{i+1}
12:         end if
13:     end for
14:     /* d_m is the bottom of the cache */
15:     if p_M < 0 then
16:         remove d_M from the list
17:     end if
18:     insert x at the front of the list
19:     return ∞
20: else
21:     /* x is a hit */
22:     find out d_k = x
23:     for i = 1; i < k; i++ do
24:         if p_i < p_{i+1} then
25:             swap d_i and d_{i+1}
26:         end if
27:     end for
28:     move x to the front of the list
29:     return k
30: end if
```

For access x at time t, Algorithm 1 computes the stack distance and updates the priority list. The algorithm has three parts:

- The first part, lines 1 to 5, sets the priority for x to be t or $-t$ depending on whether x is LRU or MRU. The purpose is to handle mixed priority. By negating t, the priority of MRU data is reverse to the access order. The MRU in the access order becomes LRU in the priority order. In addition, the negative priority means that all MRU data has a lower priority than every LRU data. Finally, all priority numbers remain distinct. As a result, all data in the cache are prioritized with no ties.
- The second part, lines 9 to 19, handles cache replacement at a miss when x is not in the priority list. The element with the lowest priority is shifted down to the bottom. It is removed if its priority is negative (an MRU datum). x is inserted to become the new head of the list.
- The third part, lines 22 to 29, handles a hit at location k, that is, $d_k = x$. The element of the lowest priority in $d_1, \ldots, d_k$ is shifted down to replace d_k. x is added to the front of the list as in the second part.

The update process, swapping and then insertion, is similar to Mattson et el. [23] but with two notable qualities. First, the priority list of bi-sim is not completely sorted, and the victim may or may not be d_M or d_k. In comparison, the priority list in LRU simulation is always totally sorted, and the victim can always be found at d_M or d_k. Second, bi-sim may remove an element from the priority list (line 16), even if it is simulating cache of an infinite size. The stack simulation of previous caching methods such as LRU and OPT never removes elements (when simulating for all cache sizes).

An example An example depicting bi-sim in action is given in Table 1. The access trace and the access types are listed in the second and third columns. The priority list (after each access) is shown in the next column. The last column is the stack distance returned by Algorithm 1: ∞ always means a miss, and k means a cache hit if cache size $C \geq k$ and a miss otherwise. The priority lists in the table show only the priority numbers p_x. A reader can find the datum from the p_xth row of the table (the p_xth access in the trace).

The example shows two notable characteristics of the bi-sim algorithm. The priority list is not completely sorted because of the negative priority numbers of MRU accesses. An MRU element may be removed from cache even when there is space, as happened at access 2. These are necessary to measure the miss ratios of all cache sizes in a single pass.

The cost and its reduction The asymptotic cost of Algorithm 1 is $O(M)$ in time and space for each access, where M is the number of distinct data elements in the input trace. The main time overhead comes from the two swap loops at lines 9-13 and 23-27. To improve performance, we divide the priority list into partially sorted groups. For example, there are 4 windows at the 25th access in the example in Table 1: [25], [21, -22], [19, -22,-23], and [16,13,10, 0] [1]. The swap loops are changed to iterate over the groups. The minimal element of a group is simply the last element. Grouping in priority lists was first invented by Sugumar and Abraham for simulating OPT [30]. A difference between OPT and bi-sim is that the accessed datum can be in the middle of a group in bi-sim. For OPT, the accessed datum always stays at the front of a group.

3.4 The Equivalence Proof

So far we have presented the bipartite cache and its simulation. We now show that the simulation algorithm is correct, that is, the

[1] For convenience, the top element is always put into a separate window.

access no.	the access trace	LRU?	the priority list (top → bottom)									stack distance
0	h	y	0									∞
1	f	n	-1	0								∞
2	i	y	2	0								∞
3	i	n	-3	0								1
4	c	y	4	0								∞
5	b	y	5	4	0							∞
6	b	n	-6	4	0							1
7	e	n	-7	4	0							∞
8	d	n	-8	4	0							∞
9	b	y	9	4	0							∞
10	g	y	10	9	4	0						∞
11	b	y	11	10	4	0						2
12	e	y	12	11	10	4	0					∞
13	d	y	13	12	11	10	4	0				∞
14	a	y	14	13	12	11	10	4	0			∞
15	c	y	15	14	13	12	11	10	0			6
16	e	y	16	15	14	13	11	10	0			4
17	a	y	17	16	15	13	11	10	0			3
18	c	y	18	17	16	13	11	10	0			3
19	i	y	19	18	17	16	13	11	10	0		∞
20	f	y	20	19	18	17	16	13	11	10	0	∞
21	b	y	21	20	19	18	17	16	13	10	0	7
22	a	n	-22	21	20	19	18	16	13	10	0	5
23	f	n	-23	21	-22	19	18	16	13	10	0	3
24	c	n	-24	21	-22	19	-23	16	13	10	0	5
25	c	y	25	21	-22	19	-23	16	13	10	0	1
26	e	n	-26	25	21	19	-22	-23	13	10	0	6
27	i	n	-27	25	21	-26	-22	-23	13	10	0	4
28	c	y	28	-27	21	-26	-22	-23	13	10	0	2
29	f	y	29	28	21	-26	-22	-27	13	10	0	6

Table 1. Example one-pass simulation of bipartite cache

elements of the priority list $d_1, d_2, \ldots, d_C$ in the algorithm are indeed the content of the LRU-MRU cache of size C. We show the equivalence in two steps. First, we show that the algorithm observes the inclusion property. Then we show that the two are equivalent at each cache size.

Proving the inclusion property is easier for the algorithm than for bipartite cache because we can use its algorithmic design directly. We first define a property in cache replacement. Let two caches of size $s, s+1$ be C_s, C_{s+1}. Assume that C_s, C_{s+1} are filled with data, and z is the element in C_{s+1} but not in C_s. At a cache miss, C_s evicts element y_s, and C_{s+1} evicts y_{s+1}. The *eviction invariance* is a property that requires

$$y_{s+1} = y_s \vee y_{s+1} = z$$

Mattson et al. [23] showed the following result:

LEMMA 2. *Eviction invariance is a necessary and sufficient condition for maintaining the inclusion property.*

Proof First, we show the necessity. If $y_{s+1} \neq y_s \wedge y_{s+1} \neq z$, y_{s+1} must be in C_s. Its eviction would mean that $C_s \notin C_{s+1}$ and would break the inclusion property. The property is also sufficient. At each eviction, if $y_{s+1} = y_s$, we have $C_{s+1} = C_s + z$; otherwise, we have $y_{s+1} = z$ and $C_{s+1} = C_s + y_s$. In both cases, $C_s \subseteq C_{s+1}$. ∎

The simulation algorithm observes the eviction invariance. The "stack" is embodied in a priority list. Each element has a numerical priority distinct from others. Therefore, the caches it simulates have the inclusion property.

LEMMA 3. *Algorithm 1 observes the eviction invariance and is therefore a stack algorithm.*

Proof Algorithm 1 identifies a victim for replacement using one of the two swap loops at lines 9-13 and 23-27. Consider two caches C_s, C_{s+1} of sizes $s, s+1$. Let z be the element in C_{s+1} but not in C_s. Let y be the element in C_s that has the lowest priority. When a cache replacement is needed in C_{s+1}, the swap loops would choose as the victim y if $p_y < p_z$ and z otherwise. The eviction invariance is therefore observed. ∎

Intuitively, the simulation is a stack algorithm because the simulated caches of all sizes share a single priority list. It is obvious that sharing a priority list implies eviction invariance. Next we show that Algorithm 1 computes the right stack distance. First we have the following lemma. We omit the proof, which is straightforward based on the handling of LRU and MRU accesses.

LEMMA 4. *At a miss in bipartite cache, the victim is always the data with the lowest priority.*

THEOREM 2. *Given an execution on bipartite cache of size C, an access is a cache hit if and only if the stack distance returned by Algorithm 1 is no greater than C.*

Proof The case for infinite distances is easy to verify, we only prove the case when the distance is of a finite value. Specifically, Algorithm 1 always stores the data in the priority list such that a cache of size C would contain and only contain the first C elements in the list, $d_1, d_2, \cdots, d_C$. This is equivalent to showing that for each data d_i, we have $d_i \in C_i$ and $d_i \notin C_{i-1}$, where $i > 0$ and C_i, C_{i-1} are the sets of data in caches of sizes $i, i-1$ respectively.

Let the memory trace be $(x_1, x_2, \cdots, x_n)$. We prove by induction on x_j.

I After accessing x_1, x_1 becomes d_1 in the priority list. The base case holds since $d_1 \in C_1$ and $d_1 \notin C_0$.

II Assume the theorem holds after accessing x_j $(1 \le j \le n-1)$. Let the element at position d_i be d_i^j and the content of caches of size $i-1, i$ be C_{i-1}^j, C_i^j. From the inductive hypothesis, we have $d_i^j \in C_i^j$ and $d_i^j \notin C_{i-1}^j$. There are two cases after accessing x_{j+1}:

(a) x_{j+1} is a (compulsory) miss. Each element of the priority list is updated from d_i^j to d_i^{j+1} $(1 \le i \le M$ or $1 \le i \le M+1)$.

i. $d_1^{j+1} = x_{j+1}$ and satisfies $d_1^{j+1} \in C_1^{j+1}$ and $d_1^{j+1} \notin C_0^{j+1}$.

ii. For d_i^{j+1} $(2 \le i \le M)$, the swap loop (lines 9 to 13) moves the datum d_h^j $(1 \le h \le i)$ with the lowest priority in C_i^j out of the priority list. According to Lemma 4, after evicting d_h from C_i^j, the top i elements in the priority list are still in C_i^{j+1}, so $d_i^{j+1} \in C_i^{j+1}$. In the same way, we can show that d_i^{j+1} is either d_i^j or the victim (of C_{i-1}^j), so $d_i^{j+1} \notin C_{i-1}^{j+1}$.

iii. If d_M^j has a positive priority, d_{M+1}^{j+1} is at the new bottom and must be the victim of C_i^j, so $d_{M+1}^{j+1} \notin C_M^{j+1}$. $d_{M+1}^{j+1} \in C_{M+1}^{j+1}$ follows from Lemma 1.

iv. If d_M^j has a negative priority, the stack distance would be infinite. It is a miss in all finite-size bipartite cache.

(b) x_{j+1} is a hit. Let the hit location be $d_k^j = x_{j+1}$. Each element of the priority list is updated from d_i^j to d_i^{j+1} $(1 \le i \le M)$.

i. Consider d_i^{j+1} $(1 \le i \le k-1)$. The access is a miss in caches $C_1^j, \cdots, C_{k-1}^j$, so the inference of the (previous) miss case can be reused here. The swap loop in lines 23 to 27 is identical to the swap loop in lines 9 to 13.

ii. Consider d_k^{j+1}. Since $C_k^j = C_k^{j+1}$, we have $d_k^{j+1} \in C_k^{j+1}$. From the inference of the miss case, $d_k^{j+1} \notin C_{k-1}^{j+1}$.

iii. Finally consider d_i^{j+1} $(k+1 \le i \le M)$, $d_i^{j+1} = d_i^j$ since there is no change made by the algorithm. From $x_{j+1} = d_k^j \in C_k^j$, we have x_{j+1} is a cache hit in C_i^j $(i \ge k+1)$ and $C_i^j = C_i^{j+1}$ $(k+1 \le i \le M)$. From the induction assumption, we have $d_i^{j+1} \in C_i^{j+1}$ and $d_i^{j+1} \notin C_{i-1}^{j+1}$ $(k+1 \le i \le M)$.

For all accesses, the cache of size C would contain and only contain the first C elements in the priority list, $d_1, d_2, \cdots, d_C$. Hence the relation is established between the stack distance and the cache hit/miss as stated in the theorem. ∎

4. Potential of Collaborative Caching

PACMAN uses OPT training analysis to select MRU memory references in a program. We first show a simple design and then use it to evaluate the potential benefit of collaborative caching.

4.1 PACMAN Design

PACMAN is a feedback-based compiler technique. As a study of the performance potential rather than a practical solution, we analyze one execution of a target program on bipartite cache of one size. The first step is OPT training, which uses an efficient OPT implementation to identify MRU accesses at the trace level and the program instructions that make these accesses. When an eviction happens in the OPT simulation, the most recent access to the victim is MRU [17]. After training, each reference in program code has a unique indicator values—the *MRU ratio*. An MRU ratio of y means that y fraction of its accesses were selected as MRU in the optimal solution. We use a simple heuristic to select MRU references: a reference is MRU if at least half of its accesses were MRU in the training run.

Once a reference is selected, all its accesses in execution will be MRU. This is most likely suboptimal. For example, if the MRU ratio of a reference is 50%, the reference will be selected and half of the accesses will be issued as MRU while they should be normal (LRU) accesses. Other heuristics may be used. Regardless of the selection method, bipartite cache will always observe the inclusion property as we have shown.

4.2 Experimental Setup

The PACMAN tool is implemented as follows. We use the gold plugin of LLVM 2.8 [1] with -O4 option to generate executables. To collect memory accesses, a profiling pass is added at the end of the link-time optimization (LTO) passes. The OPT cache simulation uses the OPT* algorithm presented in [17]. The same profiling pass is used to measure the performance of LRU and OPT caches using the fastest analyzers available [30, 37].

We examined the floating-point code in three benchmark suites—SciMark 2.0, SPEC 2000, and SPEC 2006 [2–4]—and selected those for which we can reduce the input size so the numbers of accesses are in tens of millions. We increased the number of time steps in SOR to reduce the effect of its initialization code. As mentioned earlier, as a feasibility study, we use the same input size and cache size in training and in testing. We will relax these two restrictions later.

The ten test programs are listed in Table 2. As the table shows, the programs have between 51 to 37,313 lines of C/Fortran code. There are between 12 to 10,746 static references in the programs. The length of their executions is between 100 and 800 million accesses.

We simulate fully associative bipartite cache with 8-byte cache blocks. An actual cache is always set associative, but the set associativity on modern systems is high: 4-way L1D, 10-way L2, and 12-way L3 on IBM Power 5; 8-way L1D and L2 and 16-way L3 on Intel Nehalem; and 4-way L1D and 16-way L2 on Niagara II. Hill and Smith showed that for sequential code, 8-way associative cache incurs about 5% more misses than fully-associative cache, consistently across cache sizes and cache block sizes [18]. We use fully-associative cache, so the results represent the effect of set-associative cache without being specific to particular cache parameters. As a limit study, we use 8-byte cache-line size to exclude the effect of cache spatial reuse, which depends on data layout in additional to cache management. The OPT result is the best possible (but possibly not realizable) for all data layouts.

4.3 The LRU-OPT Gap

Let $miss_{LRU}(C), miss_{OPT}(C)$ be the number of cache misses incurred by LRU and OPT cache of size C. We define the LRU-OPT gap as:

$$gap(C) = \frac{miss_{LRU}(C) - miss_{OPT}(C)}{miss_{LRU}(C)}$$

The gap is between 0 and 100%. We have simulated the LRU-OPT gap for the ten test programs for cache sizes from 1KB up to program data size (before all misses are cold-start misses). The results are summarized in Table 3.

The 2nd column of the table shows the average LRU-OPT gap for all measured cache sizes. The highest average gaps are 34% in *lucas* and 31% in *mgrid* and *milc*. The first two have hierarchical computations. The least gaps are 12% in *zeusmp* and 17% in *applu*. Both are computational fluid dynamics simulation

workload name	benchmark suite	programming language	#lines of source code	#memory references	#run-time accesses
SOR	SciMark 2.0	C	51	12	1.07E+7
171.swim	CPU2000	Fortran	435	307	1.02E+7
172.mgrid	CPU2000	Fortran	489	451	4.13E+7
173.applu	CPU2000	Fortran	3980	2515	1.50E+7
183.equake	CPU2000	C	1513	853	8.12E+7
189.lucas	CPU2000	Fortran	2999	1419	4.26E+7
410.bwaves	CPU2006	Fortran	918	755	5.30E+7
433.milc	CPU2006	C	15042	4163	8.43E+7
434.zeusmp	CPU2006	Fortran	37313	10746	3.75E+7
437.leslie3d	CPU2006	Fortran	3807	4403	3.80E+7

Table 2. The ten test programs

	the OPT imprv. over LRU		the PACMAN imprv. over LRU	
	average	largest	average	largest
SOR	25%	91%	15%	91%
171.swim	19%	64%	12%	59%
172.mgrid	31%	60%	13%	46%
173.applu	17%	50%	8.2%	19%
183.equake	22%	54%	17%	54%
189.lucas	34%	67%	26%	64%
410.bwaves	25%	80%	12%	60%
433.milc	31%	62%	8.8%	22%
434.zeusmp	12%	79%	1.4%	3.9%
437.leslie3d	27%	50%	10%	29%
average	24%	66%	12%	45%

Table 3. The LRU-OPT gap and the PACMAN improvement. The average improvement is the arithmetic mean of the improvement for all cache sizes between 1KB and data size.

programs. Across all ten programs, OPT incurs on average 24% fewer misses than LRU does on every cache size.

The improvement from LRU to OPT is not uniform. The gap can be much larger at some cache sizes. The 3rd column of the table shows that the best improvement is between 50% and 91% in all programs. In other words, for every program there is a cache size for which at least half of the misses in LRU cache can be eliminated by optimal caching. These results show a significant potential for improving cache utilization.

4.4 The Improvement by PACMAN

Let $miss_{PACMAN}(C)$ be the number of cache misses incurred by a program after the PACMAN transformation. We define the PACMAN improvement as:

$$\frac{miss_{LRU}(C) - miss_{PACMAN}(C)}{miss_{LRU}(C)}$$

The improvement may be negative if the number of misses is increased by PACMAN. We have measured the improvement for the ten test programs for all cache sizes from 1KB to the program data size. The results are in Table 3.

The 4th column of the table shows the average improvement for each program by PACMAN. Seven programs, *SOR*, *lucas*, *equake*, *mgrid*, *swim*, *bwaves*, and *leslie3d*, show 10% or more average improvements across all cache sizes. Two programs, *milc* and *applu*, show near 8% average improvements. The remaining one, *zeusmp*, does not show a significant improvement (1.4%).

The effect of PACMAN can be plotted for all cache sizes using a miss ratio curve. In this section, we show the plots first for *lucas* and *zeusmp*, which have the most and the least improvement in our test set by PACMAN; and then for *SOR*, *swim*, and *applu* to show

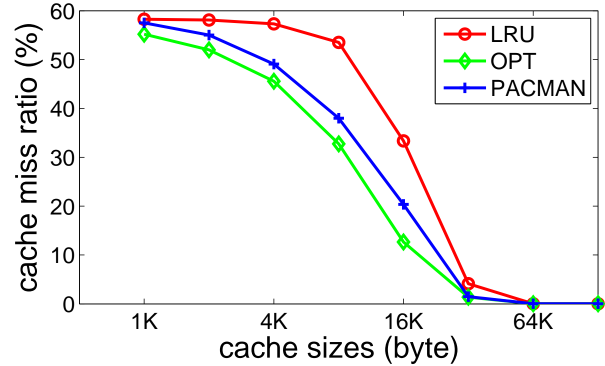


Figure 5. The miss curves of 189.lucas on fully-associative caches

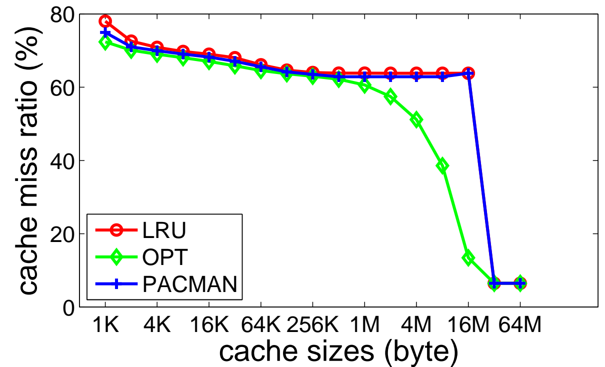


Figure 6. The miss curves of 434.zeusmp on fully-associative caches

the effects of data size, MRU ratio threshold, and cache line size. The same type graphs for the other 5 programs are included in the appendix.

Three miss ratio curves are shown in Figure 5 for *lucas* when executed with LRU caching, OPT caching, and collaborative caching. The differences between LRU and OPT curves show a large potential for improvement, on average 34% and up to 67%. The collaborative caching by PACMAN realizes over two-thirds of the potential, reducing the miss ratio by 26% on average and up to 64% in 32KB size cache.

The miss ratio curves of *zeusmp* are shown in Figure 6. There is a significant room for improvement over LRU, 12% on average

and up to 79%. While PACMAN reduces the miss ratio for almost all cache sizes, the reduction is very small (1.4% on average).

The PACMAN performance for other programs is somewhere between *lucas* and *zeusmp*, as shown by the summary in Table 3. On average, PACMAN reduces the miss ratio by 12% for each program and each cache size. Optimal caching reduces the miss ratio by 24% on average. Hence, under idealized conditions used in this study, PACMAN realizes one half of the improvement potential of optimal caching.

4.5 The Effect of Program Input

So far we train and test PACMAN on the same input. A comprehensive study on the effect of input is outside the scope of this paper (our concern here is mainly the theoretical properties and the potential). But we show that for at least one program, PACMAN shows similar improvment with different input sizes. In *swim*, the matrix size determines the program data size. We compare the results of the matrix sizes 128×128 and 256×256.

The miss ratio curves of the two executions of *swim* are shown in Figure 7. The PACMAN curve has an identical shape in both graphs, showing identical improvements over LRU. But because of the difference in input size, the improvements happen for different cache sizes—4KB, 8KB, 512KB, and 1MB for the smaller input and 8KB, 16KB, 2MB, and 4MB for the larger input. An LRU curve shows the size of working sets in an execution. Comparing the two LRU curves, we can see two working sets in this program. The first working set doubles in size in the larger input, and the second working set quadruples in size. PACMAN improves the two working sets by the same degree regardless of the input size.

4.6 The Impact of the MRU Ratio Threshold

Currently PACMAN sets the MRU ratio threshold to 50%. A program reference is designated as MRU if 50% of its run-time accesses are MRU in the optimal solution. The benefit of PACMAN depends on the choice of the threshold. Figure 9 shows one example, *173.applu* on a 512KB cache, for which different threshold values have a significant effect on performance. When the threshold is 0, all memory references become MRU. The miss ratio jumps to 99%. When the threshold is 100%, only memory references without LRU accesses are selected as MRU. The effect on this program is very close to LRU. When the threshold is 50%, the improvement over LRU is 4.6% (shown for all cache sizes in Figure 8). By choosing the threshold 30% or 35%, the miss ratio is further reduced to 20%. This suggests a higher potential if PACMAN can properly use different threshold values for different programs.

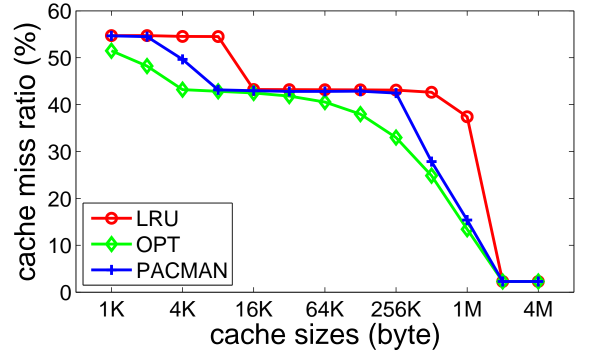
4.7 A Closer Look at SOR

All the previous evaluations are based on 8-byte cache line size for limit study. However, real cache systems usually use much larger cache line size such as 64-byte. We change to use 64-byte cache line size to make a more realistic test with SOR.

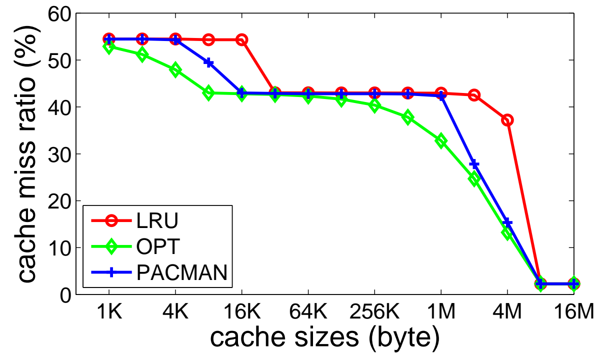
Figure 10 shows the SSA-form [13] of the SOR loop kernel (for original code see Figure 2). The loop indexes into array G to create three virtual arrays $Gi, Gim1, Gip1$ for use in the innermost loop.

The innermost loop has 4 array references. The MRU ratio changes with cache sizes, as shown by Figure 11 as a curve for each reference. The ratio for $Gim1[j]$ is clearly higher than the other three. For cache size between 8KB and 512KB, the MRU ratio is from over 12.5% to 63% for $Gim1[j]$ but near 0 for the other three. PACMAN chooses this reference as an MRU reference.

The MRU ratio is a factor of 8 lower because of spatial reuse. To separate the last touch of a cache block, we transform line 9 to an if-else block (line 9.1 to 9.5) in Figure 10. In actual implementation, we use loop unrolling instead of branching. In LLVM, we adapt the available loop unrolling pass and put it at the end of the LTO passes



(a) matrix size is 128*128



(b) matrix size is 256*256

Figure 7. The miss curves of 171.swim on two different inputs. The curves have an identical shape but cover different cache-size ranges: between 1KB and 4MB in the upper graph and between 1KB and 16MB in the lower graph.

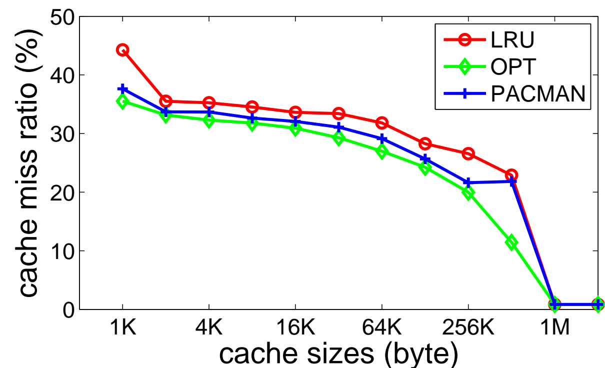


Figure 8. The miss curves of 173.applu on fully-associative caches

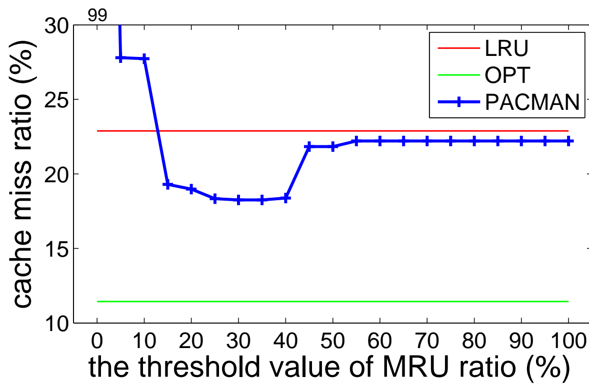


Figure 9. The impact of the MRU ratio threshold for 173.applu at 512KB

```
Require: G is a 2-dimensional double array with the size M*N
 1: for p = 1; p < NUM_STEPS; p++ do
 2:   for i = 1; i < M-1; i++ do
 3:     Gi = G[i];
 4:     Gim1 = G[i-1];
 5:     Gip1 = G[i+1];
 6:     Gijm1 = Gi[0];
 7:     Gij = Gi[1];
 8:     for j = 1; j < N-1; j++ do
 9:       Gim1j = Gim1[j];      ==>   9.1: if j%8 == 7 then
10:       Gip1j = Gip1[j];            9.2:   Gim1j =
11:       Gijp1 = Gi[j+1];                       MRU_load(Gim1[j]);
12:       tmp1 = Gim1j + Gip1j;       9.3: else
13:       tmp1 += Gijm1;              9.4:   Gim1j = Gim1[j];
14:       tmp1 += Gijp1;              9.5: end if
15:       tmp1 *= 0.3125;
16:       tmp2 = -0.25 * Gij;
17:       tmp1 += tmp2;
18:       Gi[j] = tmp1;
19:       Gijm1 = tmp1;
20:       Gij = Gijp1;
21:     end for
22:   end for
23: end for
```

Figure 10. The SOR kernel loop in SSA form with PACMAN transformation. `M = N = 512` and `NUM_STEPS = 10`.

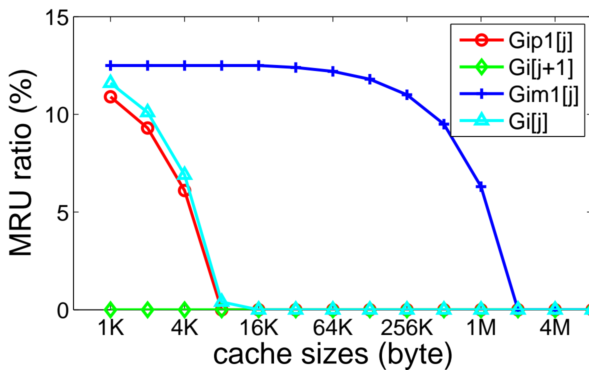


Figure 11. The MRU ratio curves of SOR on fully-associative caches with cache line size 64B

but before the profiling pass. We also change to use `memalign()` instead of `malloc()` to make array `G` 64-byte aligned. After loop unrolling, the load in line 9.2 has an MRU ratio of 75% at 512KB.

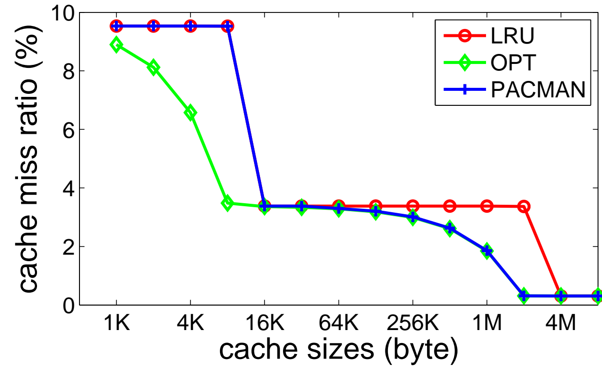


Figure 12. The miss curves of SOR on fully-associative caches with cache line size 64B

The miss ratio curves of Figure 12 show that PACMAN produces almost identical results as OPT for cache sizes over 64KB (up to 2MB). The improvements are significant—2.3%, 5.2%, 10.8%, 22.2%, 44.9%, and 90.7% respectively between 64KB to 2MB. The average improvement is 15%, as reported in Table 3. It is worth mentioning that at cache size 2MB, OPT training found 704 MRU accesses out of more than ten million accesses. These MRU accesses reduced the miss ratio by an order of magnitude from 3.3% to 0.3%.

5. Related Work

Cache hints The ISA of Intel Itanium extends the interface of the memory instruction to provide source and target hints [5]. The source hint suggests where data is expected, and the target hint suggests which level cache the data should be kept. The target hint changes the cache replacement decisions in hardware. IBM Power processors support bypass memory access that do not keep the accessed data in cache [27]. Wang et al. proposed an interface to tag cache data with evict-me bits [31]. Recently, Ding et al. developed ULCC which uses page coloring to partition cache to separately store high locality and low locality data [15]. It may be used to approximate LRU-MRU cache management in software on existing machines. The bipartite cache interface in this paper can imitate the effect of the target hints, cache bypasses, and evict-me bits. Consequently, the theoretical properties such as the inclusion principle and bipartite stack distance are valid for these existing designs of collaborative cache.

Collaborative caching Collaborative caching was pioneered by Wang et al. [31] and Beyls and D'Hollander [8, 9]. The studies were based on a common idea, which is to evict data whose forward reuse distance is larger than the cache size. Wang et al. used compiler analysis to identify self and group reuse in loops [24, 31, 32] and select array references to tag with the evict-me bit. They showed that collaborative caching can be combined with prefetching to further improve performance.

Beyls and D'Hollander used profiling analysis to measure the reuse distance distribution for each program reference. They added cache hint specifiers on Intel Itanium and improved average performance by 10% for scientific code and 4% for integer code [8]. Profiling analysis is input specific. Fang et al. showed a technique that accurately predicts how the reuse distances of a memory reference change across inputs [16]. Beyls and D'Hollander later developed static analysis called reuse-distance equations and obtained similar improvements without profiling [9]. Compiler analysis of

reuse distance was also studied by Cascaval and Padua for scientific code [10] and Chauhan and Shei for Matlab programs [11].

In this paper, we show the theoretical potential of collaborative caching. With bipartite cache, these techniques may be extended to achieve optimal cache performance. OPT analysis is a possible extension. It is more precise. Recall an example in Section 4.7 where a few hundred MRU accesses can reduce the miss ratio of a ten-million long trace by an order of magnitude. Such OPT training can be used to evaluate and improve compiler and profiling-based techniques.

Virtual machine, operating system and hardware memory management Garbage collectors may benefit from the knowledge of application working set size and the affinity between memory objects. For LRU cache, reuse distance has been used by virtual machine systems to estimate the working set size [34] and to group simultaneously used objects [36]. There have been much research in operating systems to improve beyond LRU. A number of techniques used last reuse distance instead of last access time in virtual memory management [12, 28, 38] and file caching [19]. The idea of evicting dead data early has been extensively studied in hardware cache design, including deadblock predictor [22], adaptive cache insertion [26], less reuse filter [33], virtual victim cache [21], and globalized placement [35].

Hardware based techniques improve memory and cache performance without changing software. On the flip side, they do not allow software to communicate information about its data usage. This communication is the goal of collaborative cache. We believe the idea is also interesting for heap and virtual memory management. A basic problem in collaborative systems is that the interface may be misused. We have shown the theoretical properties of this interface under all uses. Particularly important for software is that the LRU-MRU stack distance exists and may be used to estimate the working set size and reference affinity in collaborative cache as reuse distance has been used for conventional cache.

Optimal caching Optimal caching is difficult purely at the program level. Kennedy and McKinley [20] and Ding and Kennedy [14] showed that optimal loop fusion is NP hard. Petrank and Rawitz showed that given the order of data access and cache management, the problem of optimal data layout is intractable unless P=NP [25]. Our earlier workshop paper showed that collaborative caching can be used to obtain optimal cache performance [17]. It described two extensions to LRU called bypass LRU and trespass LRU and gave an counter example showing bypass LRU does not observe the inclusion principle. The paper gave an efficient algorithm for simulating OPT cache replacement, which we use in this paper for PACMAN training analysis. The previous study assumed that a program could be optimally transformed. In this paper, we study the properties of collaborative caching in all uses, not just optimal uses.

6. Summary

In this paper, we have characterized the difference between current LRU-style cache management and optimal cache management. To approximate optimal solution on real cache systems, we have formalized the interface of bipartite LRU-MRU cache and shown that it obeys the inclusion principle. We give a one-pass simulation algorithm to measure the LRU-MRU stack distance. We have measured the potential of collaborative caching using a simple algorithm based on OPT training analysis. The evaluation on 10 SciMark and SPEC CPU benchmarks show that optimal caching can reduce the miss ratio by 24% on average per program per cache size, and collaborative caching has the potential to realize 50% of the optimal performance improvement.

Acknowledgments

We wish to thank Luke K. Dalessandro for help with LLVM. We also wish to thank Tongxin Bai, Bin Bao, Arrvindh Shriraman, Xiaoya Xiang, and the anonymous reviewers for their comments and/or proofreading.

The research is supported by the National Science Foundation (Contract No. CCF-0963759, CNS- 0834566, CNS-0720796) and IBM CAS Faculty Fellowships. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the authors and do not necessarily reflect the views of the funding organizations.

References

[1] The LLVM Compiler Infrastructure. `http://llvm.org/`.

[2] SciMark2.0. `http://math.nist.gov/scimark2/`.

[3] SPEC CPU2000. `http://www.spec.org/cpu2000`.

[4] SPEC CPU2006. `http://www.spec.org/cpu2006`.

[5] *IA-64 Application Developer's Architecture Guide*. May 1999.

[6] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 1966.

[7] L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Communications of ACM*, 1969.

[8] K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, 2002.

[9] K. Beyls and E. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 2005.

[10] C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *International Conference on Supercomputing*, 2003.

[11] A. Chauhan and C.-Y. Shei. Static reuse distances for locality-based optimizations in MATLAB. In *International Conference on Supercomputing*, 2010.

[12] F. Chen, S. Jiang, and X. Zhang. CLOCK-Pro: an effective improvement of the CLOCK replacement. In *Proceedings of USENIX Annual Technical Conference*, 2005.

[13] R. Cytron, J. Ferrante, B. K. Rosen, M. N. Wegman, and F. K. Zadeck. Efficiently computing static single assignment form and the control dependence graph. *ACM Transactions on Programming Languages and Systems*, 1991.

[14] C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 2004.

[15] X. Ding, K. Wang, and X. Zhang. ULCC: a user-level facility for optimizing shared cache performance on multicores. In *Proceedings of the ACM SIGPLAN Symposium on Principles and Practice of Parallel Programming*, 2011.

[16] C. Fang, S. Carr, S. Önder, and Z. Wang. Instruction based memory distance analysis and its application. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2005.

[17] X. Gu, T. Bai, Y. Gao, C. Zhang, R. Archambault, and C. Ding. P-OPT: Program-directed optimal cache management. In *Proceedings of the Workshop on Languages and Compilers for Parallel Computing*, 2008.

[18] M. D. Hill and A. J. Smith. Evaluating associativity in CPU caches. *IEEE Transactions on Computers*, 1989.

[19] S. Jiang and X. Zhang. LIRS: an efficient low inter-reference recency set replacement to improve buffer cache performance. In *Proceedings of the International Conference on Measurement and Modeling of Computer Systems*, 2002.

[20] K. Kennedy and K. S. McKinley. Typed fusion with applications to parallel and sequential code generation. Technical Report TR93-208, Dept. of Computer Science, Rice University, 1993.

[21] S. M. Khan, D. A. Jiménez, D. Burger, and B. Falsafi. Using dead blocks as a virtual victim cache. In *Proceedings of the 19th international conference on Parallel architectures and compilation techniques*, 2010.

[22] A.-C. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the International*

Symposium on Computer Architecture, 2001.

[23] R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 1970.

[24] K. S. McKinley, S. Carr, and C.-W. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 1996.

[25] E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of the ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, 2002.

[26] M. K. Qureshi, A. Jaleel, Y. N. Patt, S. C. S. Jr., and J. S. Emer. Adaptive insertion policies for high performance caching. In *Proceedings of the International Symposium on Computer Architecture*, 2007.

[27] B. Sinharoy, R. N. Kalla, J. M. Tendler, R. J. Eickemeyer, and J. B. Joyner. Power5 system microarchitecture. *IBM J. Res. Dev.*, 2005.

[28] Y. Smaragdakis, S. Kaplan, and P. Wilson. The EELRU adaptive replacement algorithm. *Perform. Eval.*, 2003.

[29] K. So and R. N. Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, 1988.

[30] R. A. Sugumar and S. G. Abraham. Efficient simulation of caches under optimal replacement with applications to miss characterization. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, 1993.

[31] Z. Wang, K. S. McKinley, A. L.Rosenberg, and C. C. Weems. Using the compiler to improve cache replacement decisions. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, 2002.

[32] M. E. Wolf and M. Lam. A data locality optimizing algorithm. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, 1991.

[33] L. Xiang, T. Chen, Q. Shi, and W. Hu. Less reused filter: improving L2 cache performance via filtering less reused lines. In *Proceedings of the 23rd international conference on Supercomputing*, 2009.

[34] T. Yang, E. D. Berger, S. F. Kaplan, and J. E. B. Moss. Cramm: Virtual memory support for garbage-collected applications. In *Proceedings of the Symposium on Operating Systems Design and Implementation*, 2006.

[35] M. Zahran and S. A. McKee. Global management of cache hierarchies. In *Proceedings of the 7th ACM international conference on Computing frontiers*, 2010.

[36] C. Zhang and M. Hirzel. Online phase-adaptive data layout selection. In *Proceedings of the European Conference on Object-Oriented Programming*, 2008.

[37] Y. Zhong, X. Shen, and C. Ding. Program locality analysis using reuse distance. *ACM Transactions on Programming Languages and Systems*, 2009.

[38] P. Zhou, V. Pandey, J. Sundaresan, A. Raghuraman, Y. Zhou, and S. Kumar. Dynamic tracking of page miss ratio curve for memory management. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, 2004.

A. The Miss Ratio Curves of LRU, OPT, and PACMAN

Figure 13 to 17 shows the miss-ratio curves of the rest of the test programs (in addition to programs already shown in Section 4).

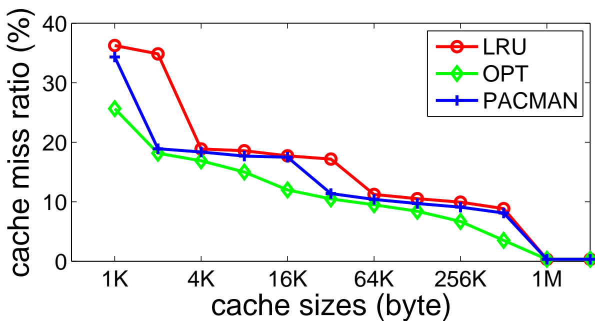


Figure 13. 172.mgrid

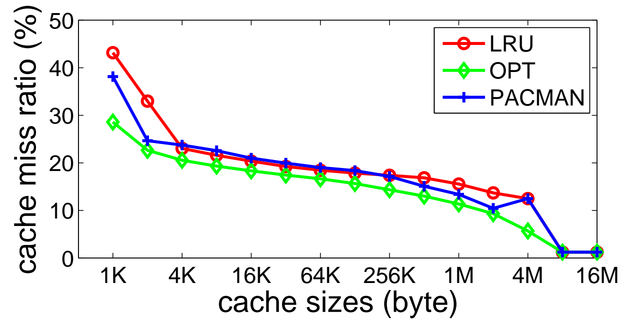


Figure 14. 183.equake

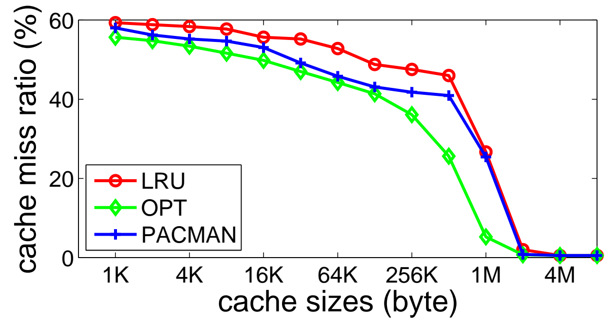


Figure 15. 410.bwaves

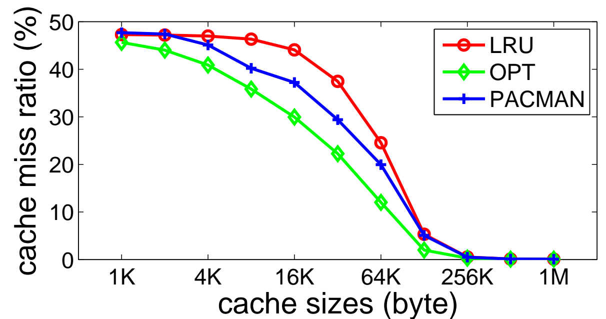


Figure 16. 433.milc

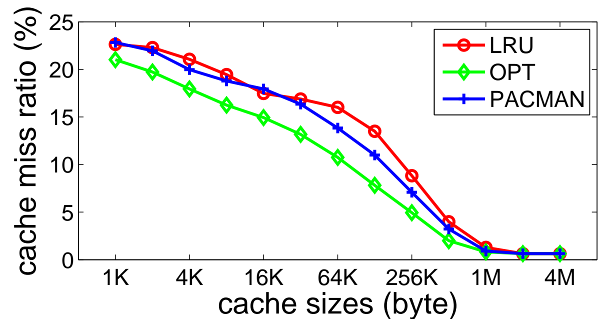


Figure 17. 437.leslie3d