On the Theory and Potential of Collaborative Cache Management

Xiaoming Gu       Chen Ding
{xiaoming,cding}@cs.rochester.edu

The University of Rochester
Computer Science Department
Rochester, NY 14627

Technical Report TR-965
February 2011

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 cache 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 38% of the optimal improvement.

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.
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 (Kennedy and McKinley, 1993; Petrank and Rawitz, 2002; Ding and Kennedy, 2004).

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 (Beyls and D’Hollander, 2005), bypassing access on IBM Power series (Sinharoy et al., 2005), and evict-me bit (Wang et al., 2002). Our earlier work showed a theoretical result that two extensions of LRU cache may be managed optimally by a program (Gu et al., 2008). Wang et al. called a combined software-hardware solution collaborative caching (Wang et al., 2002).

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 (Wang et al., 2002), 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, 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 higher level cache is contained in the caches below (Mattson et al., 1970). The inclusion principle has important benefits. In theory, the miss ratio is a monotone function of cache size. There is no Belady anomaly (Belady et al., 1969). 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 (Mattson et al., 1970). 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 (Yang et al., 2006; Zhang and Hirzel, 2008).

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 (Mattson et al., 1970).
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 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, cache is 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 (Hill and Smith, 1989). 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 (So and Rechtschaffen, 1988). 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 (Zhong et al., 2009).

**OPT** The *optimal replacement algorithm* (OPT) was invented by Belady (Belady, 1966). 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 (Mattson et al., 1970). Sugumar and Abraham invented an efficient one-pass algorithm (Sugumar and Abraham, 1993). 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 algorithms** 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* (Mattson et al., 1970). The miss rate is a monotone (non-increasing) function of cache size, and there is no Belady’s anomaly (Belady et al., 1969). The paper proved the inclusion property for LRU, LFU (least frequently used), OPT, and a form of random replacement. A practical value of the
inclusion property is that we can measure the miss rate for cache of all sizes in a single pass without having to simulate each cache size separately.

**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 (NIST, 2000). We use currently the fastest one-pass analysis methods for LRU (Ding and Zhong, 2003) and OPT (Sugumar and Abraham, 1993). 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 to 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.

**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 \( M \times N \) 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.

### 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.
Figure 1: The gap between LRU and OPT in SOR

**Require:** G is a 2-dimensional double array with the size M*N

```plaintext
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

### 3.1 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 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 (Gu et al.,
(a) A LRU hit: \( w \), assuming at entry \( S_3 \), is moved to the top of the stack

\[
S_1 \quad S_2 \quad S_3(w) \quad \Rightarrow \quad S_1 \quad S_2 \quad S_3(w)
\]

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

\[
w \quad \Rightarrow \quad w \quad S_m
\]

Figure 3: A LRU memory access

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

\[
S_1 \quad S_2 \quad S_3(w) \quad \Rightarrow \quad S_1 \quad S_2 \quad S_3(w)
\]

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

\[
w \quad S_m \quad \Rightarrow \quad w \quad S_m
\]

Figure 4: An MRU memory access
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 significant ways:

- **Bipartite content.** The cache stack is divided into two parts: the upper part for LRU data and the lower part for MRU data.

- **Capacity dependent placement.** The MRU access is placed at the bottom location, which is determined by the size of the cache.

- **Two-way priority.** The LRU part of the cache 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, elements in LRU cache are ordered by the last access time, and elements in OPT cache are ordered by the next access time. The ordering is uniform and does not depend on cache size. The uniform ordering naturally gives rise to the inclusion property and its practical benefits. To understand collaborative caching, we must understand the consequence of its bipartite nature.

### 3.2 The Inclusion Property

We prove that for any sequence of LRU and MRU accesses, the bipartite cache obeys the inclusion principle. The property is beneficial in evaluation, when we can evaluate all cache sizes in one simulation, and in analysis, when we can target an abstract stack distance.

If the collaborative cache is used optimally, the performance is the same as OPT (Gu et al., 2008). 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 (Mattson et al., 1970).

**Lemma 1** If the bottom element in the bipartite cache stack is last visited by a normal access, then all elements in cache are last visited by normal 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 (Gu et al., 2008). Next we prove the inclusion property.

**Theorem 1** A trace $P$ is being executed on two bipartite caches of sizes $C_1, C_2, C_1 < C_2$. 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 \( C_1, C_2 \) 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 \( C_1, C_2 \) contain \( x_p, x_q \) but in an opposite order.

The two accesses, \( x_p, x_q \), 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 \( C_2 \supseteq C_1 \), 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 One-pass Simulation of Bipartite cache

We call the algorithm for bipartite cache simulation bi-sim in short. To understand bi-sim, 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

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

Mattson et al. (Mattson et al., 1970) 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 $y_{s+1} \neq y_s \land y_{s+1} \neq z$, $y_{s+1}$ 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 bipartite cache simulation is designed to observe eviction invariance. The pseudo-code is given in Algorithm 1. The “stack” is embodied in a priority list. Each element has a numerical priority distinct from others. Note that the list is not fully sorted.

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 7 to 16, 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 18 to 24, 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 al. (Mattson et al., 1970) 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 13), 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
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. $\forall i, j, p_i \neq p_j$. 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: if $x \notin \{d_i : i = 1, \ldots, M\}$ then \{x is a miss\}
7: \quad for $j = 1; j < M; j++$ do
8: \quad \quad if $p_j < p_{j+1}$ then
9: \quad \quad \quad swap $d_j$ and $d_{j+1}$
10: \quad \quad end if
11: \quad end for
12: if $p_M < 0$ then
13: \quad remove $d_M$ from the list
14: end if
15: insert $x$ at the front of the list
16: return $\infty$
17: else \{x is a hit and $x = d_k$\}
18: \quad for $j = 1; j < k; j++$ do
19: \quad \quad if $p_j < p_{j+1}$ then
20: \quad \quad \quad swap $d_j$ and $d_{j+1}$
21: \quad \quad end if
22: \quad end for
23: insert $x$ at the front of the list
24: return $k$
25: end if
(each access) is shown in the next column. The last column is the stack distance returned by Algorithm 1: $\infty$ 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 $p_x$th row of the table (the $p_x$th 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.

**Correctness** We prove the inclusion property of the algorithm.

**Theorem 2** 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 7-11 and 18-22. Consider two caches of sizes $C_s, C_{s+1}$. If the victim chosen by the swap loop is not in $C_{s+1}$, the content of $C_s, C_{s+1}$ does not change, so is their inclusion relation. Now assume that the victim is from $C_{s+1}$. 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. Then the swap loops would choose as the victim $y$ if $p_y < p_z$ and $z$ otherwise. The eviction invariance is therefore observed.

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 swap loops in the algorithm.

**Lemma 3** At a miss in bipartite cache, the victim is always the data with the lowest priority.

**Theorem 3** 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 cache of size $C$ would contain and only contain the first $C$ elements in the list, $d_1, d_2, \ldots, d_C$. 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 $(x_1, x_2, \ldots, x_n)$. 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 $x_j$ ($1 \leq j \leq n-1$). 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 $x_{j+1}$:
<table>
<thead>
<tr>
<th>access no.</th>
<th>the access trace</th>
<th>MRU or not</th>
<th>the priority list (top → bottom)</th>
<th>stack distance</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>h</td>
<td>n</td>
<td>0</td>
<td>∞</td>
</tr>
<tr>
<td>1</td>
<td>f</td>
<td>y</td>
<td>-1 0</td>
<td>∞</td>
</tr>
<tr>
<td>2</td>
<td>i</td>
<td>n</td>
<td>2 0</td>
<td>∞</td>
</tr>
<tr>
<td>3</td>
<td>i</td>
<td>y</td>
<td>-3 0</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>c</td>
<td>n</td>
<td>4 0</td>
<td>∞</td>
</tr>
<tr>
<td>5</td>
<td>b</td>
<td>n</td>
<td>5 4 0</td>
<td>∞</td>
</tr>
<tr>
<td>6</td>
<td>b</td>
<td>y</td>
<td>-6 4 0</td>
<td>1</td>
</tr>
<tr>
<td>7</td>
<td>e</td>
<td>y</td>
<td>-7 4 0</td>
<td>∞</td>
</tr>
<tr>
<td>8</td>
<td>d</td>
<td>y</td>
<td>-8 4 0</td>
<td>∞</td>
</tr>
<tr>
<td>9</td>
<td>b</td>
<td>n</td>
<td>9 4 0</td>
<td>∞</td>
</tr>
<tr>
<td>10</td>
<td>g</td>
<td>n</td>
<td>10 9 4 0</td>
<td>∞</td>
</tr>
<tr>
<td>11</td>
<td>b</td>
<td>n</td>
<td>11 10 4 0</td>
<td>2</td>
</tr>
<tr>
<td>12</td>
<td>e</td>
<td>n</td>
<td>12 11 10 4 0</td>
<td>∞</td>
</tr>
<tr>
<td>13</td>
<td>d</td>
<td>n</td>
<td>13 12 11 10 4 0</td>
<td>∞</td>
</tr>
<tr>
<td>14</td>
<td>a</td>
<td>n</td>
<td>14 13 12 11 10 4 0</td>
<td>∞</td>
</tr>
<tr>
<td>15</td>
<td>c</td>
<td>n</td>
<td>15 14 13 12 11 10 0</td>
<td>6</td>
</tr>
<tr>
<td>16</td>
<td>e</td>
<td>n</td>
<td>16 15 14 13 11 10 0</td>
<td>4</td>
</tr>
<tr>
<td>17</td>
<td>a</td>
<td>n</td>
<td>17 16 15 13 11 10 0</td>
<td>3</td>
</tr>
<tr>
<td>18</td>
<td>c</td>
<td>n</td>
<td>18 17 16 13 11 10 0</td>
<td>3</td>
</tr>
<tr>
<td>19</td>
<td>i</td>
<td>n</td>
<td>19 18 17 16 13 11 10 0</td>
<td>3</td>
</tr>
<tr>
<td>20</td>
<td>f</td>
<td>n</td>
<td>20 19 18 17 16 13 11 10 0</td>
<td>∞</td>
</tr>
<tr>
<td>21</td>
<td>b</td>
<td>n</td>
<td>21 20 19 18 17 16 13 10 0</td>
<td>7</td>
</tr>
<tr>
<td>22</td>
<td>a</td>
<td>y</td>
<td>-22 21 20 19 18 16 13 10 0</td>
<td>5</td>
</tr>
<tr>
<td>23</td>
<td>f</td>
<td>y</td>
<td>-23 21 -22 19 18 16 13 10 0</td>
<td>3</td>
</tr>
<tr>
<td>24</td>
<td>c</td>
<td>y</td>
<td>-24 21 -22 19 -23 16 13 10 0</td>
<td>5</td>
</tr>
<tr>
<td>25</td>
<td>c</td>
<td>n</td>
<td>25 21 -22 19 -23 16 13 10 0</td>
<td>1</td>
</tr>
<tr>
<td>26</td>
<td>e</td>
<td>y</td>
<td>-26 25 21 19 -22 -23 13 10 0</td>
<td>6</td>
</tr>
<tr>
<td>27</td>
<td>i</td>
<td>y</td>
<td>-27 25 21 -26 -22 -23 13 10 0</td>
<td>4</td>
</tr>
<tr>
<td>28</td>
<td>c</td>
<td>n</td>
<td>28 -27 21 -26 -22 -23 13 10 0</td>
<td>2</td>
</tr>
<tr>
<td>29</td>
<td>f</td>
<td>n</td>
<td>29 28 21 -26 -22 -27 13 10 0</td>
<td>6</td>
</tr>
</tbody>
</table>

Table 1: Example one-pass simulation of bipartite cache
For all accesses, the cache of size $C$ and the cache hit/miss as stated in the theorem.

- $x_{j+1}$ is a miss. Each element of the priority list is updated from $d_i^j$ to $d_i^{j+1}$ ($1 \leq i \leq M$ or $1 \leq i \leq M + 1$).
  - $d_i^{j+1} = x_{j+1}$ and satisfies $d_i^{j+1} \in C_i^{j+1}$ and $d_i^{j+1} \notin C_i^{j+1}$.
  - For $d_i^{j+1}$ ($2 \leq i \leq M$), the swap loop (lines 7 to 11) moves the datum $d_h^j$ ($1 \leq h \leq i$) of the lowest priority in $C_i^j$ out of the priority list. According to Lemma 3, 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 $d_i^{j+1} \notin C_i^{j+1}$.
  - If $d_M^{i}$ has a positive priority, $d_M^{i+1}$ is at the new bottom and must be the victim of $C_i^{j}$, so $d_M^{i+1} \notin C_M^{j+1}$. $d_M^{i+1} \in C_M^{j+1}$ follows from Lemma 1.
  - If $d_M^j$ has a negative priority, the stack distance would be infinite. It is a miss in all finite-size bipartite cache.

- $x_{j+1}$ is a hit. Let the hit location be $d_k^j = x_{j+1}$. Each element of the priority list is updated from $d_k^j$ to $d_k^{j+1}$ ($1 \leq i \leq M$).
  - Consider $d_k^{j+1}$ ($1 \leq i \leq k - 1$). The access is a miss in caches $C_i^j, \ldots, C_{k-1}^j$, so the inference of the (previous) miss case can be reused here. The swap loop in lines 18 to 22 is identical to the swap loop in lines 7 to 11.
  - 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, $d_k^{j+1} \notin C_k^{j+1}$.
  - Finally consider $d_i^{j+1}$ ($k + 1 \leq i \leq M$), $d_i^{j+1} = d_i^j$ since there is no change made by the algorithm. From $d_k \in C_i^j$ and Theorem 2, we have $x_{j+1}$ is a cache hit in $C_i$ ($i \geq k + 1$) and $C_i^j = C_i^{j+1}$ ($k + 1 \leq i \leq M$). From the induction assumption, we have $d_i^{j+1} \in C_i^{j+1}$ and $d_i^{j+1} \notin C_i^{j+1}$ ($k + 1 \leq i \leq M$).

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

**The cost and its reduction**  The asymptotic cost of Algorithm 1 is $O(M)$ in time and space, 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 7-11 and 18-22. 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].

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 (Sugumar and Abraham, 1993). 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.

---

For convenience, the top element is always put into a separate window.
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 (Gu et al., 2008) to identify MRU accesses at the trace level and the program instructions that make these accesses. After training, each reference in program code has two indicator values—the access ratio and the MRU ratio. An access ratio of \( x \) means that \( x \) fraction of the data accesses in the trace. An MRU ratio of \( y \) means that \( y \) fraction of its accesses were selected as MRU in the optimal solution.

Algorithm 2 shows a simple heuristic to select MRU references: a reference is MRU if it accounted for more than 0.1% of data accesses and at least half of its accesses were MRU in the training run.

**Algorithm 2** The MRU reference selection by PACMAN

Require: The access ratio and MRU ratio for each memory reference in the program.

1. for each reference \( ref \) in the program do
2. if access ratio \( \geq 0.001 \) and MRU ratio \( \geq 0.5 \) then
3. mark \( ref \) for MRU access
4. end if
5. end for

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 0.5, 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 (LLVM, 2010) 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 (Gu et al., 2008). The same profiling pass is used to measure the performance of LRU and OPT cache using the fastest analyzers available (Sugumar and Abraham, 1993; Ding and Zhong, 2003).

We examined the floating-point code in three benchmark suites—SciMark, SPEC 2000, and SPEC 2006 (NIST, 2000; SPEC, 2000, 2006)—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. 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 (Hill and Smith, 1989). We use fully-associative cache, so the results represent the effect set-associative cache without being specific to particular cache parameters.

### 4.3 The LRU-OPT Gap

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

\[
\text{gap}(C) = \frac{\text{miss}_{LRU}(C) - \text{miss}_{OPT}(C)}{\text{miss}_{LRU}(C)}
\]

The gap is between 0 and 100%. We have measured 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.
(a) The miss ratio curves of 189.lucas

(b) The miss ratio curves of 437.leslie3d
The 2nd column of the table shows the average LRU-OPT gap for all measured cache sizes. The highest average gaps are observed in code with hierarchical computations, 34% in lucas and 31% in mgrid. The least gaps are observed in computational fluid dynamics simulation, 12% in (the magnetohydrodynamics code) zeusmp and 17% in applu. On average across all ten programs, OPT incurs 24% fewer misses than LRU does.

The improvement 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. Five programs, SOR, lucas, mgrid, swim, bwaves, show over 10% average improvement across all cache sizes. Three programs, milc, equake, applu, show 8% and 2% average
improvements. The rest two, zeusmp, leslie3d, do not show a significant improvement (less than 1%).

The effect of PACMAN can be plotted for all cache sizes using the miss ratio curves. In this section, we show the plots first for lucas, leslie3d, which have the most and the least improvement by PACMAN; and then for SOR, swim to show the effect of data size and cache-block size. The same type graphs for the other 6 programs are included in the appendix.

Three miss ratio curves are shown in Figure 5(a) 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 32% and up to 67%. The collaborative caching by PACMAN realizes over half the potential, reducing the miss ratio by 23% on average and up to 65% in 32KB size cache.

The miss ratio curves of leslie3d are shown in Figure 5(b). There is a significant room for improvement over LRU, 27% on average and up to 50%. The current design of PACMAN causes more misses in small size caches (up to 32KB) and shows slight improvements in two larger cache sizes. The average is less than 1%.

The PACMAN performance for other programs is somewhere between lucas and leslie3d, as shown by the summary in Table 3. On average, PACMAN reduces the miss ratio by 8.8% 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 over one third (38%) of the improvement potential of optimal caching.

4.5 A Closer Look at SOR

Figure 5 shows the SSA-form (Cytron et al., 1991) of the SOR loop kernel (for original code see Figure 2). The loop indexes into array G to create three virtual arrays Gi, Gim, Gip for use in the innermost loop.

The innermost loop has 4 array references, with an identical access ratio of 0.24. The MRU ratio changes with the cache size, as shown by Figure 6(a) as a curve for each reference. The ratio for Gim[j] is clearly higher than the other three. For cache size between 8KB and 512KB, the MRU ratio is over 0.1 for Gim[j] but near 0 for the other three. PACMAN chooses this reference as an MRU reference.

In this test, we change the cache block size from 8 bytes to 64 bytes. 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 5. 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 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 access ratio 3% and an MRU ratio of 75% at 512KB.

The miss ratio curves of Figure 6(b) show that PACMAN produces identical results as OPT for cache sizes over 16KB (up to 4MB). The improvements are significant between 64KB to 2MB—2.3%, 5.2%, 10.8%, 22.2%, 44.9%, and 90.7% respectively. The average improvement is 15%, as reported in Table 3. It is worth mentioning that at cache size 2MB,
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:      Gi = G[i+1]; 9.3: else
12:      tmp1 = Gim1j + Gip1j; 9.4: Gim1j =
13:      tmp1 += Gijm1; 9.5: end if
14:      tmp1 += Gip1j;
15:      tmp1 *= 0.3125;
16:      tmp2 = -0.25 * Gij;
17:      tmp1 += tmp2;
18:      Gij = tmp1;
19:      Gijm1 = tmp1;
20:   end for
21: end for
22: end for

Figure 5: The SOR kernel loop in SSA form with PACMAN transformation. M = N = 512 and NUM_STEPS = 10.
(a) The MRU ratio curves of SOR on fully-associative cache with cache line size 64B

(b) The miss curves of SOR on fully-associative cache with cache line size 64B
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%.

4.6 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 can be trained on a small input and obtain improvement for a larger input. In *swim*, the matrix size determines program data size. We use the matrix size $128 \times 128$ in training and evaluate the program for the same matrix size and for matrix size $256 \times 256$.

The miss ratio curves of the two executions of *swim* are shown in Figure 6. The PACMAN curve has an identical shape in both graphs, showing identical improvements over LRU. But because of the difference in input size, these same improvements happen for different cache sizes—between 4KB and 1MB for the smaller input and between 8KB and 4MB for the larger input. The LRU curves show two working sets in this program. Comparing the two curves, we can see that the first working set doubles in size in the larger input, and the second working set quadruples in size. The same PACMAN solution improves performance in the same fashion for the two inputs.

5 Related Work

**Cache hints** The ISA of Intel Itanium extends the interface of the memory instruction to provide source and target hints (Intel, 1999). 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 (Sinharoy et al., 2005). Wang et al. proposed an interface to tag cache data with evict-me bits (Wang et al., 2002). 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 is pioneered by Wang et al. (Wang et al., 2002) and Beyls and D’Hollander (Beyls and D’Hollander, 2002, 2005). 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 (Wolf and Lam, 1991; McKinley et al., 1996; Wang et al., 2002) 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 (Beyls
Figure 6: 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 left-hand side graph and between 1KB and 16MB in the right-hand side graph.
and D’Hollander, 2002). They later developed static analysis called reuse-distance equations and obtained similar improvements without profiling (Beyls and D’Hollander, 2005). Compiler analysis of reuse distance was also studied by Cascaval and Padua for scientific code (Cascaval and Padua, 2003) and Chauhan and Shei for Matlab programs (Chauhan and Shei, 2010).

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 4.5 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 (Yang et al., 2006) and to group simultaneously used objects (Zhang and Hirzel, 2008). 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 (Smaragdakis et al., 2003; Zhou et al., 2004; Chen et al., 2005) and file caching (Jiang and Zhang, 2002). The idea of evicting dead data early has been extensively studied in hardware cache design, including deadblock predictor (Lai et al., 2001), adaptive cache insertion (Qureshi et al., 2007), less reuse filter (Xiang et al., 2009), and virtual victim cache (Khan et al., 2010).

VM, OS, or 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 data 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 (Kennedy and McKinley, 1993) and Ding and Kennedy (Ding and Kennedy, 2004) 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 (Petrank and Rawitz, 2002). Our earlier workshop paper showed that collaborative caching can be used to obtain optimal cache performance (Gu et al., 2008). It describes two extensions to LRU called bypass LRU and trespass LRU and gives an counter example showing bypass LRU does not observe the inclusion principle. The paper gives 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 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 38% 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, and Xiaoya Xiang for their comments and/or proof-reading.

Appendix: The Miss Ratio Curves of LRU, OPT, and PACMAN

Figure 7 to 12 shows the miss-ratio curves of the rest of the test programs (in addition to programs already shown in Section 4).

References

Intel. 1999. IA-64 Application Developer’s Architecture Guide.


Figure 7: The miss curves of 172.mgrid on fully-associative cache

Figure 8: The miss curves of 173.applu on fully-associative cache
Figure 9: The miss curves of 183.equake on fully-associative cache

Figure 10: The miss curves of 410.bwaves on fully-associative cache
Figure 11: The miss curves of 433.milc on fully-associative cache

Figure 12: The miss curves of 434.zeusmp on fully-associative cache