P-OPT: Program-directed Optimal Cache Management*

Xiaoming Gu[o], Tongxin Bai[†], Yaoqing Gao[‡], Chengliang Zhang[^], Roch Archambault[‡], and Chen Ding[†]

[o]Intel China Research Center, Beijing, China
[†]Department of Computer Science, University of Rochester, New York, USA
[‡]IBM Toronto Software Lab, Ontario, Canada
[^] Microsoft Redmond Campus, Washington, USA
{xiaoming,bai,zhangchl,cding}@cs.rochester.edu
{ygao,archie}@ca.ibm.com

Abstract. As the amount of on-chip cache increases as a result of Moore's law, cache utilization is increasingly important as the number of processor cores multiply and the contention for memory bandwidth becomes more severe. Optimal cache management requires knowing the future access sequence and being able to communicate this information to hardware. The paper addresses the communication problem with two new optimal algorithms for *Program-directed OPTimal cache management (P-OPT)*, in which a program designates certain accesses as bypasses and trespasses through an extended hardware interface to effect optimal cache utilization. The paper proves the optimality of the new methods, examines their theoretical properties, and shows the potential benefit using a simulation study and a simple test on a multi-core, multi-processor PC.

1 Introduction

Memory bandwidth has become a principal performance bottleneck on modern chip multi-processors because of the increasing contention for off-chip data channels. Unlike the problem of memory latency, the bandwidth limitation cannot be alleviated by data prefetching or multi-threading. The primary solution is to minimize the cache miss rate. Optimal caching is NP-hard if we consider computation and data reorganization [8, 12]. If we fix the computation order and the data layout, the best caching is given by the optimal replacement strategy, MIN [2]. Since MIN cannot be implemented purely in hardware, today's machines use variations of LRU (least recently used) and random replacement. This leaves room for significant improvement—LRU can be worse than optimal by a factor proportional to the cache size in theory [14] and by a hundred folds in practice [6].

* The research was conducted while Xiaoming Gu and Chengliang Zhang were graduate students at the University of Rochester. It was supported by two IBM CAS fellowships and NSF grants CNS-0720796, CNS-0509270, and CCR-0238176.

Recent architectural designs have added an interface for code generation at compile time to influence the hardware cache management at run time. Beyls and D'Hollander used the cache-hint instructions available on Intel Itanium to specify which level of cache to load a data block into [5]. Wang et al. studied the use of the evict-me bit, which, if set, informs the hardware to replace the block in the cache first when space is needed [15]. The techniques improve cache utilization by preserving the useful data in cache either explicitly through cache hints or implicitly through the eviction of other data. The advent of collaborative cache management raises the question of whether or not optimal cache management is now within reach.

In the paper, we will prove that in the ideal case where the operation of each access can be individually specified, two simple extensions to the LRU management can produce optimal results. We will first discuss the optimal algorithm MIN and its stack implementation OPT. We then describe a new, more efficient implementation of OPT called OPT* and the two LRU extensions, *bypass LRU* and *trespass LRU*, that use OPT* in off-line training and to generate annotated traces for the two LRU extensions. We will show an interesting theoretical difference that bypass LRU is not a stack algorithm but trespass LRU is. Finally, we will demonstrate the feasibility of program annotation for bypass LRU and the potential improvement on a multi-core, multi-processor PC.

2 Two new optimal cache management algorithms

In this paper an *access* means a memory operation (load/store) at run time and a *reference* means a memory instruction (load/store) in the executable binary. An access is a *hit* or *miss*, depending whether the visited data element is in cache immediately before the access.

The operation of an access has three parts: the *placement* of the visited element, the *replacement* of an existing element if the cache is full, and the *shift* of the positions or priorities of the other elements. The shift may or may not be an actual action in the hardware, depending on implementation.

Following the classic model of Mattson et al. [11], we view cache as a stack or an ordered array. The data element at the top of the array has the highest priority and should be the last to evict, and the data element at the bottom is the next to evict when space is needed.

The original MIN solution by Belady is costly to implement because it requires forward scanning to find the cache element that has the furthest reuse [2]. Noting this problem, Mattson et al. described a two-pass stack algorithm, which computes the forward reuse distance in the first pass and then in the second pass maintains a priority list based on the pre-computed forward reuse distance [11]. Mattson et al. gave a formal proof the optimality in the 7-page appendix through four theorems and three lemmata. They called it the OPT algorithm. The main cost of OPT is the placement operation, which requires inserting an element into a sorted list. In comparison, the cost of LRU is constant. It places the visited element at the top of the LRU stack, which we call the *Most Recently Used (MRU)*

Policies	Placement Cost	Shift Cost	Replacement Cost	Optimal?	Stack Alg?
MIN	constant	none	$O(N+M)$ for forward scanning plus selection	Yes	Yes
OPT	$O(M)$ list insertion	$O(M)$ update	constant	Yes	Yes
LRU	constant	none	constant	No	Yes
OPT*	$O(\log M)$ list insertion	none	constant	Yes	Yes
bypass LRU	constant	none	constant	Yes	No
trespass LRU	constant	none	constant	Yes	Yes

Table 1. The time complexity of cache-management algorithms in terms of the cost per access for placement, shift, and replacement operations. N is the length of the trace, and M is the size of cache.

position, and it evicts the bottom element, which we call the *Least Recently Used (LRU)* position.

Table 1 compares six cache-management algorithms mentioned in this paper, where M is the cache size and N is the length of the access sequence. The first three rows show the cost of MIN, OPT, and LRU, and the next three rows show the algorithms we are to present: OPT*, bypass LRU, and trespass LRU. It shows, for example, that the original OPT algorithm requires an update cost of $O(M)$ per data access, but OPT* needs no such update and that bypass and trespass LRU have the same cost as LRU. Note that the cost is for the on-line management. The two LRU extensions require running OPT* in the training analysis. All but LRU can achieve optimal cache utilization. All but bypass LRU are stack algorithms.

Bypass and Trespass LRU algorithms use training analysis to specify the type of each cache access. Next we describe the three types of cache access and the OPT* algorithm used in training.

2.1 Three Types of Cache Access

We describe the normal LRU access and define the hardware extensions for bypass LRU access and trespass LRU access.

- *Normal LRU access* uses the most-recently used position for placement and the least-recently used position for replacement
  - Miss: Evict the data element S_m at LRU position (bottom of the stack) if the cache is full, shift other data elements down by one position, and place w, the visited element, in the MRU position (top of the stack). See Figure 1.
  - Hit: Find w in cache, shift the elements over w down by one position, and re-insert w at the MRU position. See Figure 2. Note that search cost is constant in associative cache where hardware checks all entries in parallel.
- *Bypass LRU access* uses the LRU position for placement and the same position for replacement. It is similar to the bypass instruction in IA64 [1] except that its bypass demotes the visited element to LRU position when hit.

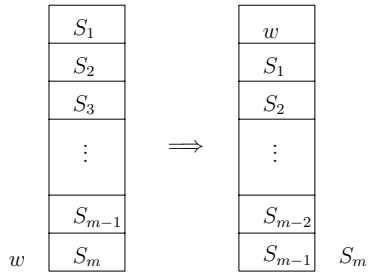

Fig. 1. Normal LRU at a miss: w is placed at the top of the stack, evicting S_m

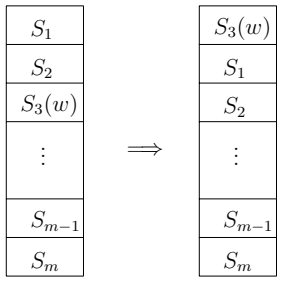

Fig. 2. Normal LRU at hit: w, assuming at entry S_3, is moved to the top of the stack

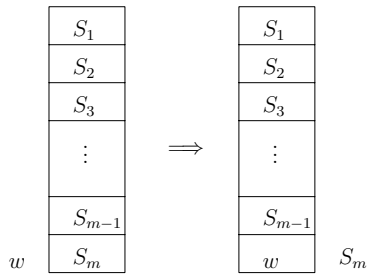

Fig. 3. Bypass LRU at a miss: the bypass posits w at the bottom of the stack, evicting S_m

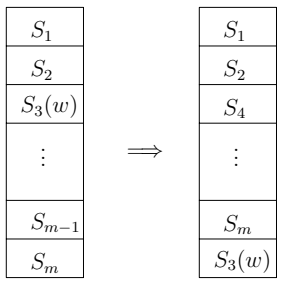

Fig. 4. Bypass LRU at a hit: the bypass moves $S_3(w)$ to the bottom of the stack

  - Miss: Evict S_m at the LRU position if the cache is full and insert w into the LRU position. See Figure 3.
  - Hit: Find w, lift the elements under w by one position, and place w in the LRU position. See Figure 4.
- *Trespass LRU access* uses the most-recently used position for placement and the same position for replacement. It differs from all cache replacement policies that we are aware of in that both the cache insertion and eviction happen at one end of the LRU stack.
  - Miss: Evict the data element S_1 at the MRU position if the cache is not empty and insert w in the MRU position. See Figure 5.
  - Hit: If w is in the MRU position, then do nothing. Otherwise, evict the data element S_1 at the MRU position, insert w there, and shift the elements under the old w spot up by one position. See Figure 6.

2.2 OPT* Algorithm

Given a memory access sequence, the original OPT algorithm has two passes [11]:

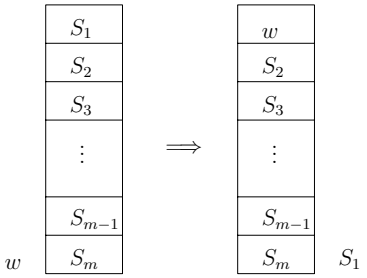

Fig. 5. Trespass LRU at a miss: the trespass posits w at the top of the stack, evicting S_1

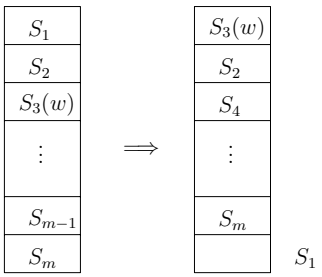

Fig. 6. Trespass LRU at a hit: the trespass raises $S_3(w)$ to the top of the stack, evicting S_1

– First pass: Compute the forward reuse distance for each access through a backward scan of the trace.
– Second pass: Incrementally maintain a priority list based on the forward reuse distance of the cache elements. The pass has two steps. First, if the visited element is not in cache, find its place in the sorted list based on its forward reuse distance. Second, after each access, update the forward reuse distance of each cache element.

The update operation is costly and unnecessary. To maintain the priority list, it is sufficient to use the next access time instead of the forward reuse distance. At each point p in the trace, the next access time of data x is the logical time of the next access of x after p. Since the next access time of data x changes only at each access of x, OPT* stores a single next access time at each access in the trace, which is the next access time of the element being accessed. OPT* collects next access times through a single pass traversal of the trace. The revised algorithm OPT* is as follows.

– First pass: Store the next reuse time for each access through a backward scan of the trace.
– Second pass: Maintaining the priority list based on the next reuse time. It has a single step. If the visited element is not in cache, find its place in the sorted list based on its next access time.

The cost per operation is $O(\log M)$ for cache of size M, if the priority list is maintained as a heap. It is asymptotically more efficient than the $O(M)$ per access cost of OPT. The difference is computationally significant when the cache is large. While OPT* is still costly, it is used only for pre-processing and adds no burden to on-line cache management.

2.3 The Bypass LRU Algorithm

In bypass LRU, an access can be a normal access or a bypass access, which are described in Section 2.1. The type of each access is determined using OPT* in

the training step. To ensure optimality, the trace of the actual execution is the same as the trace used in training analysis. For each miss in OPT*, let d be the element evicted and x be the last access of d before the eviction, the training step would tag x as a bypass access. After training, the untagged accesses are normal accesses.

The training result is specific to the cache size being used. We conjecture that the dependence on cache size is unavoidable for any LRU style cache to effect optimal caching. The result is portable, in the sense that the performance does not degrade if an implementation optimized for one cache size is used on a machine with a larger cache. A compiler may generate code for a conservative size at each cache level or generate different versions of the code for different cache sizes for some critical parts if not whole application. Finally, the training for different cache sizes can be made and the access type specified for each cache level in a single pass using OPT*.

Two examples of bypass LRU are shown in Table 2 to demonstrate the case where the cache is managed with the same constant cost per access as LRU, yet the result is optimal, as in OPT*.

Bypass LRU is not a stack algorithm This is shown using a counter example. By comparing the two sub-tables in Table 2, we see that at the first access to e, the stack content, given in bold letters, is (**e,d**) in the smaller cache and (**e, c, b**) in the larger cache. Hence the inclusion property does not hold and bypass LRU is not a stack algorithm [11].

(a) cache size = 2

Trace	a	b	c	d	d	c	e	b	e	c	d
Bypasses	X	X			X	X			X	X	
Bypass LRU Stack	a	b		c	d	c	**d**	e	e	c	b
Misses	1	2	3	4			5	6		7	8
OPT* Stack	a	b	c	d	d	c	e	b	e	c	d
		a	b	c	c	d	c	e	b	b	b
Misses	1	2	3	4			5	6		7	8

(b) cache size = 3

Trace	a	b	c	d	d	c	e	b	e	c	d
Bypasses	X				X					X	
Bypass LRU Stack		b	c	d	c	c	**e**	b	e	e	d
			b	c	b	b	**c**	e	b	b	e
	a			b	d	d	**b**	c	c	c	b
Misses	1	2	3	4			5				6
OPT* Stack	a	b	c	d	d	c	e	b	e	c	d
		a	b	c	c	d	c	e	b	e	e
			a	b	b	b	b	c	c	b	b
Misses	1	2	3	4			5				6

Table 2. Two examples showing bypass LRU is optimal but is not a stack algorithm

Bypass LRU is optimal In Figure 2, bypass LRU has the same number of cache misses as OPT*, which is optimal. We next prove the optimality for all traces.

Lemma 1 *If the bottom element in the bypass LRU stack is last visited by a normal access, then all cache elements are last visited by some normal accesses.*

Proof. If some data elements are last visited by bypass accesses, then they appear only at the bottom of the stack. They can occupy multiple positions but cannot be lifted up over an element last visited by a normal access. Therefore, if the bottom element is last visited by a normal access, all elements in the cache must also be. ∎

Theorem 1 *Bypass LRU generates no more misses than OPT*. In particular, bypass LRU has a miss only if OPT* has a miss.*

Proof. We show that there is no access that is a cache hit in OPT* but a miss in Bypass LRU. Suppose the contrary is true. Let $z\prime$ be the first access in the trace that hits in OPT* but misses in Bypass LRU. Let d be the element accessed by $z\prime$, z be the immediate previous access to d, and the reference trace between them be $(z, ..., z\prime)$.

The access z can be one of the two cases.

- z is a normal access. For z' to miss in bypass LRU, there should be a miss y in $(z, ..., z\prime)$ that evicts d. From the assumption that $z\prime$ is the earliest access that is a miss in bypass LRU but a hit in OPT*, y must be a miss in OPT*. Consider the two possible cases of y.
  - y occurs when the OPT* cache is partially full. Since the OPT* cache is always full after the loading of the first M elements, where M is the cache size, this case can happen only at the beginning. However, when the cache is not full, OPT* will not evict any element. Hence this case is impossible.
  - y occurs when the OPT* cache is full. The element d is at the LRU position before the access of y. According to Lemma 1, the bypass LRU cache is full and the last accesses of all data elements in cache are normal accesses. Let the set of elements in cache be T for bypass LRU and T^* for OPT*. At this time (before y), the two sets must be identical. The reason is a bit tricky. If there is an element d' in the bypass LRU cache but not in the OPT* cache, d' must be replaced by OPT* before y. However, by the construction of the algorithm, the previous access of d' before y should be labeled a bypass access. This contradicts to the lemma, which says the last access of d' (and all other elements in T) is normal. Since both caches are full, they must be identical, so we have $T = T^*$. Finally, y in the case of OPT* must evict some element. However, evicting any element other than d would violate our lemma. Hence, such y cannot exist and this case is impossible.
- z is a bypass access in Bypass LRU. There must be an access $y \in (z, ..., z\prime)$ in the case of OPT* that evicts d; otherwise z cannot be designated as a bypass. However, in this case, the next access of d, $z\prime$ cannot be a cache hit in OPT*, contradicting the assumption that $z\prime$ is a cache hit in OPT*.

Considering both cases, it is impossible for the same access to be a hit in OPT* but a miss in bypass LRU. ∎

Since OPT* is optimal, we have the immediate corollary that bypass LRU has the same number of misses as OPT* and is therefore optimal. In fact, the misses happen for the same accesses in bypass LRU and in OPT*. Last, we show that Bypass LRU as a cache management algorithm has a peculiar feature.

Corollary 1 *Although Bypass LRU is not a stack algorithm, it does not suffer from Belady anomaly [3], in which the number of misses sometimes increases when the cache size becomes larger.*

Proof. OPT is a stack algorithm since the stack content for a smaller cache is a subset of the stack content for a larger cache [11]. The number of misses of an access trace does not increase with the cache size. Since bypass LRU has the same number of misses as OPT*, it has the same number of misses as OPT and does not suffer from Belady anomaly. ∎

2.4 The Trespass LRU Algorithm

In trespass LRU, an access can be a normal access or a trespass access. The two obvious choices for efficient LRU stack replacement are evicting from the bottom, as in bypass LRU just described, or evicting from the top, as in trespass LRU. Both are equally efficient at least asymptotically. We will follow a similar approach to show the optimality of trespass LRU. The main proof is actually simpler. We then show an unexpected theoretical result—trespass LRU is a stack algorithm, even though bypass LRU is not.

Similar to bypass LRU, trespass LRU uses a training step based on simulating OPT* for the given cache on the given trace. For each miss y in OPT*, let d be the evicted cache element and x be the last access of d before y. The training step tags the access immediately after x as a trespass access. It is trivial to show that such an access exists and is unique for every eviction in OPT*.

Two example executions of trespass LRU execution are shown in Table 3 for the same trace used to demonstrate bypass LRU in Table 2. .

Trespass LRU is Optimal The effect of a trespass access is less direct than that of a bypass access. We need four additional lemmata. First, from the way trespass accesses are identified, we have

Lemma 2 *If a data element w is evicted by a trespass access x, then x happens immediately after the last access of w.*

Lemma 3 *If a data element is in trespass LRU cache at point p in the trace, then the element is also in OPT* cache at p.*

Proof. Assume that a data element w is in the trespass LRU cache but is evicted from the OPT* cache. Let x be the last access of w. Consider the time of the eviction in both cases. The eviction by trespass LRU happens right after x. Since the eviction by OPT* cannot be earlier, there must be no period of time when an element w is in the trespass LRU cache but not in the OPT* cache. ∎

(a) cache size = 2

Trace	a	b	c	d	d	c	e	b	e	c	d
Trespasses		X	X			X	X			X	X
Trespass LRU Stack	a	b	c	d	d	c	e	b	e	c	d
				c	c			e	b	b	b
Misses	1	2	3	4			5	6		7	8
OPT* Stack	a	b	c	d	d	c	e	b	e	c	d
		a	b	c	c	d	c	e	b	b	b
Misses	1	2	3	4			5	6		7	8

(b) cache size = 3

Trace	a	b	c	d	d	c	e	b	e	c	d
Trespasses		X				X					X
Trespass LRU Stack	a	b	c	d	d	c	e	b	e	c	d
			b	c	c	b	c	e	b	e	e
				b	b		b	c	c	b	b
Misses	1	2	3	4			5				6
OPT* Stack	a	b	c	d	d	c	e	b	e	c	d
		a	b	c	c	d	c	e	b	e	e
			a	b	b	b	b	c	c	b	b
Misses	1	2	3	4			5				6

Table 3. Two examples showing trespass LRU is optimal and is a stack algorithm (unlike bypass LRU)

Lemma 4 *If a data element is evicted by a normal access in trespass LRU, then the cache is full before the access.*

This is obviously true since the normal access cannot evict any element unless the cache is full. Not as obvious, we have the following

Lemma 5 *A normal access cannot evict a data element from cache in trespass LRU.*

Proof. Assume y is a normal access that evicts data w. Let T and T^* be the set of data elements in the Trespass LRU cache and the OPT* cache before access y. By Lemma 3, $T \subseteq T^*$. By Lemma 4, the Trespass LRU cache is full before y. Then we have $T = T^*$. In OPT*, y has to evict some element $d \in T^*$. Let x be the last access of d before y. Since Trespass LRU evicts d right after x, the content of the cache, T and T^* cannot be the same unless y is the next access after x, in which case, d is w, and y must be a trespass access. ∎

Theorem 2 *Trespass LRU generates no more misses than OPT*. In particular, trespass LRU has a miss only if OPT* has a miss.*

Proof. We show that there is no access that is a cache hit in OPT* but a miss in trespass LRU. Suppose the contrary is true. Let $z\prime$ be the first access in the trace that hits in OPT* but misses in Trespass LRU. Let d be the element accessed by $z\prime$, z be the immediate previous access to d, and the reference trace between them be $(z, ..., y, ..., z\prime)$, where y is the access that causes the eviction of d in trespass LRU.

By Lemma 5, y is a trespass access. By Lemma 2, y happens immediately after z. Since y is a trespass after z, then the next access of d, $z\prime$ must be a miss in OPT*. This contradicts the assumption that $z\prime$ is a hit in OPT*. Therefore, any access that is a miss in trespass LRU must also be a miss in OPT*. ∎

Corollary 2 *Trespass LRU has the same number of misses as OPT* and is therefore optimal.*

Trespass LRU is a stack algorithm Given that bypass LRU is not a stack algorithm, the next result is a surprise and shows an important theoretical difference between trespass LRU and bypass LRU.

Theorem 3 *Trespass LRU is a stack algorithm.*

Proof. Assume there are two caches C_1 and C_2. C_2 is larger than C_1, and the access sequence is $Q = (x_1, x_2, ..., x_n)$. Let $T_1(t)$ be the set of elements in cache C_1 after access x_t and $T_2(t)$ be the set of elements in cache C_2 after the same access x_t. The initial sets for C_1 and C_2 are $T_1(0)$ and $T_2(0)$, which are empty and satisfy the inclusion property. We now prove the theorem by induction on t.

Assume $T_1(t) \subseteq T_2(t)$ $(1 \leq t \leq n-1)$. There are four possible cases based on the type of the access x_{t+1} when visiting either of the two caches. We denote the data element accessed at time x_i as $D(x_i)$.

- If x_{t+1} is a trespass access both in C_1 and C_2, we have

$$\begin{aligned} T_1(t+1) &= T_1(t) - D(x_t) + D(x_{t+1}) \\ &\subseteq T_2(t) - D(x_t) + D(x_{t+1}) \\ &= T_2(t+1) \end{aligned}$$

- If x_{t+1} is a trespass access in C_1 but a normal access in C_2, then by Lemma 5, x_{t+1} does not cause any eviction in cache C_2 and therefore

$$\begin{aligned} T_1(t+1) &= T_1(t) - D(x_t) + D(x_{t+1}) \\ &\subseteq T_2(t) + D(x_{t+1}) \\ &= T_2(t+1) \end{aligned}$$

- The case that x_{t+1} is a normal access in C_1 but a trespass access in C_2 is impossible. Since x_{t+1} is a trespass in C_2, $D(x_t)$ would be evicted by some access y in C_2 using OPT*. However, x_{t+1} is a normal access in C_1, which means that $D(x_t)$ is in C_1 after access y when using OPT*. This in turn means that at the point of y, the inclusion property of OPT* no longer holds and contradicts the fact that OPT* is a stack algorithm.
- If x_{t+1} is a normal access both in C_1 and C_2, then by Lemma 5, x_{t+1} does not cause an eviction either in C_1 or C_2, and therefore

$$\begin{aligned} T_1(t+1) &= T_1(t) + D(x_{t+1}) \\ &\subseteq T_2(t) + D(x_{t+1}) \\ &= T_2(t+1) \end{aligned}$$

From the induction hypothesis, the inclusion property holds for Trespass LRU for all t. ∎

The next corollary follows from the stack property.

Corollary 3 *Trespass LRU as a cache management algorithm does not suffer from Belady anomaly [3].*

In Table 3, we have revisited the same data trace used to show that bypass LRU was not a stack algorithm. It shows that the inclusion property holds when trespass LRU is used. The example also shows that trespass LRU cache can become partially empty after it becomes full. Trespass LRU keeps the visited data element and the data elements that will be visited. When the amount of data that have a future reuse is less than the cache size, OPT* and bypass LRU may contain extra data elements that have no future reuse. In OPT* the extra data do not destroy the inclusion property, but in bypass LRU they do.

2.5 Limitations

Bypass LRU and trespass LRU solve the problem of efficient on-line cache management, but their optimality depends on specifying the type of individual accesses, which is not feasible. In practice, however, a compiler can use transformations such as loop splitting to create different memory references for different types of accesses. We will show this through an example in the next section.

Another problem is the size of program data may change in different executions. We can use the techniques for predicting the change of locality as a function of the input [9, 16] and use transformations such as loop splitting to specify caching of only a constant part of the (variable-size) program data.

Throughout the paper, we use the fully associative cache as the target. The set associative cache can be similarly handled by considering it as not just one but a collection of fully associative sets. All the theorems about the Bypass and Trespass LRU hold for each fully associative set, and the optimality results stay the same, so are the stack properties.

The training analysis can also be extended naturally to tag bypass or trespass accesses for each set. There is an additional issue of the data layout, especially if it changes with the program input. Though we have not developed any concrete solutions, we believe that these problems can be approached by more sophisticated training, for example, pattern analysis across inputs, and additional program transformations such as loop splitting.

Bypass LRU is better than trespass LRU because the latter is sensitive to the order of accesses. It is possible that a trespass access is executed at an unintended time as a result of instruction scheduling by the compiler and the out-of-order execution by the hardware. In comparison, the effect of bypass is not sensitive to such reordering.

3 The Potential Improvements of P-OPT

3.1 A Simulation Study

While controlling the type of each access is impractical, we may use simple transformations to approximate bypass LRU at the program level based on the result of training analysis. Assume the fully associative cache has 512 blocks, and each block holds one array element. The code in Figure 7, when using

```
int a[1000]
for(j=1;j<=10;j++)
 for(i=1;i<=997;i++)
  a[i+2]=a[i-1]+a[i+1];
```

Fig. 7. Original Code

```
int a[1000]
for(j=1;j<10;j++) {
 for(i=1;i<=509;i++)
  a[i+2]=a[i-1]+a[i+1];
 for(;i<=996;i++)
  a[i+2]=a[i-1]+a[i+1];
 for(;i<=997;i++)
  a[i+2]=a[i-1]+a[i+1];
}
```

Fig. 8. Transformed Code

LRU, causes 10000 capacity misses among the total 29910 accesses. The minimal number of misses is half as much or, to be exact, 5392, given by OPT*. There are three array references in the original loop. OPT* shows that the accesses by reference a[i+1] are all normal accesses except for three accesses. The accesses by reference a[i+2] are all normal accesses. Finally reference a[i-1] has a cyclic pattern in every 997 accesses with about 509 normal accesses and 488 bypass accesses in each period.

Based on the training result, we split the loop into three parts as shown in Figure 8. In the first loop, all three references are *normal references*, which means the accesses by them are all normal accesses. In the second loop, the references a[i+1] and a[i+2] are normal references but reference a[i-1] is tagged with the *bypass bit*, which means that it is a *bypass reference* and its accesses are all bypass accesses. In the third loop, the references a[i+2] are normal references but references a[i-1] and a[i+1] are bypass references. The transformed program yields 5419 cache misses, an almost half reduction from LRU and almost to same as the optimal result of 5392 misses. After the transformation, it looks like we retain the first part of array a in cache but bypass the second part. Effectively it allocates the cache to some selected data to utilize cache more efficiently.

Not all programs may be transformed this way. Random access is an obvious one for which optimal solution is impossible. However, for regular computations, this example, although simple, demonstrates that P-OPT with training based bypass LRU may obtain near optimal cache performance.

3.2 A Simple But Real Test

We use a program which repeatedly writes to a contiguous area the size of which is controlled by the input as the data size. The access has the large stride of 256 bytes so the average memory access latency is high. The second input parameter to the program is the retainment size specifying how large piece of data should be retained in the cache. Since the access to the rest of the data takes space in cache, the retainment size is smaller than the cache size. Considering the set-associativity of the cache structure, we set the retainment size to $\frac{3}{4}$ of the total cache size. When multiple processes are used, the total retainment size is divided evenly among them. Our testing machine has two Core Duo chips, so

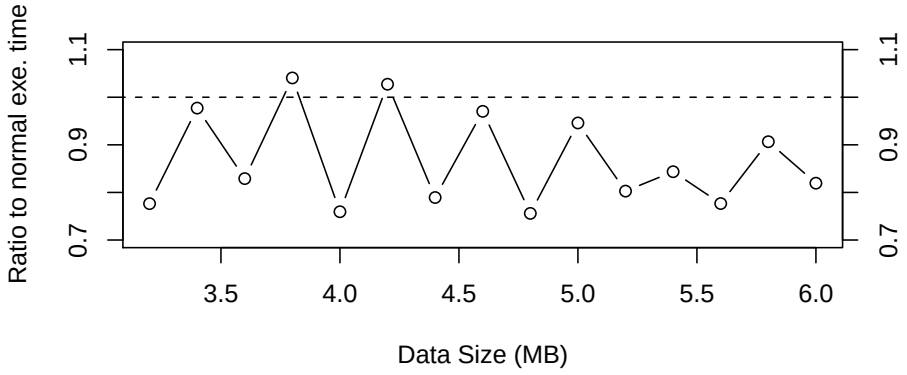

Fig. 9. Ratio of running time between using bypassing stores and using normal stores in 4 processes. The lower the ratio is, the faster the bypassing version. The machine has two Intel Core Duo processors, and each has two 3GHz cores and 4MB cache.

cache contention only happens when four processes run together. In that case, we give half of the total retainment size to each process. The running time is measured for 50 million memory accesses.

Figure 9 shows the effect of cache bypassing running four processes. The instruction `movnti` on Intel x86 processors is used to implement the cache bypassing store. With four processes and contention for memory bandwidth, the improvement is observed at 3.2MB and higher data sizes. The worst is 4% slowdown at 3.8MB, the best is 24% at 4.8MB, and the average is 13%. The result is tantalizing: the bypassing version runs with 13% less time or 15% higher speed, through purely software changes. Our store bypassing is a heuristic that may not be optimal. Therefore the potential improvement from bypass LRU is at least as great, if not greater.

4 Related Work

The classic studies in memory management [2,11] and self-organizing data structures [14] considered mostly uniform data placement strategies such as LRU, OPT, and MIN. This paper establishes a theoretical basis for selective replacement strategies that are optimal yet can be implemented on-line with the same cost as LRU. The cost of annotation is shifted to an off-line step.

The use of cache hints and cache bypassing at the program level is pioneered by two studies in 2002. Beyls and D'Holander used a training-based analysis for inserting cache hints [4]. By instructing the cache to replace cache blocks

that have long-distance reuses, their method obtained 9% average performance improvement on an Intel Itanium workstation [4]. Wang et al. published a set of compiler techniques that identified different data reuse levels in loop nests and inserted evict-me bits for references for which the reuse distance was larger than the cache size [15]. Beyls and D'Holander later developed a powerful static analysis (based on the polyhedral model and integer equations and called reuse-distance equations) and compared it with training-based analysis for scientific code [5].

Our scheme differs from the two earlier methods because bypass and trespass LRU are designed to preserve in cache data blocks that have long-distance reuses. Intuitively speaking, the goal of the previous methods is to keep the working set in the cache if it fits, while our goal is to cache a part of the working set even if it is larger than cache. At the implementation level, our method needs to split loop iterations. Beyls and D'Holander considered dynamic hints for caching working sets that fit in cache [5]. Qureshi et al. recently developed a hardware scheme that selectively evicted data based on the predicted reuse distance [13]. The study showed significant benefits without program-level inputs. As a pure run-time solution, it naturally incorporates the organization of the cache and the dynamics of an execution, but it is also inherently limited in its predictive power.

One important issue in training based analysis is the effect of data inputs. Fang et al. gave a predictive model that predicted the change of the locality of memory references as a function of the input [9]. They showed on average over 90% accuracy across program inputs for 11 floating-point and 11 integer programs from the SPEC2K CPU benchmark suite. Their result suggested that training based analysis can accurately capture and exploit the reuse patterns at the memory reference level. Marin and Mellor-Crummey demonstrated the cross-input locality patterns for larger code units such as loops and functions [10]. In addition for scientific code, compiler analysis can often uncover the reuse-distance pattern, as demonstrated by Cascaval and Padua [7] and Beyls and D'Holander [5], and eliminate the need of training analysis.

5 Summary

In this paper we have presented two new cache management methods, bypass LRU and trespass LRU, which are programmable by software, require similar hardware support as LRU, and may produce the same result as optimal cache management. Bypass LRU is not a stack algorithm while trespass LRU is. Both require training analysis, for which we have presented OPT*, asymptotically the fastest implementation of optimal cache management. We have demonstrated preliminary evidence that bypass LRU can be effectively used by a combination of off-line training and program transformation. The better utilization of cache has led to significant performance improvement for parallel memory traversals on multi-core processors.

References

1. *IA-64 Application Developer's Architecture Guide*. May 1999.
2. L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):78–101, 1966.
3. L. A. Belady, R. A. Nelson, and G. S. Shedler. An anomaly in space-time characteristics of certain programs running in a paging machine. *Commun. ACM*, 12(6):349–353, 1969.
4. K. Beyls and E. D'Hollander. Reuse distance-based cache hint selection. In *Proceedings of the 8th International Euro-Par Conference*, Paderborn, Germany, August 2002.
5. K. Beyls and E. D'Hollander. Generating cache hints for improved program efficiency. *Journal of Systems Architecture*, 51(4):223–250, 2005.
6. D. C. Burger, J. R. Goodman, and A. Kagi. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23th International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.
7. C. Cascaval and D. A. Padua. Estimating cache misses and locality using stack distances. In *Proceedings of International Conference on Supercomputing*, San Francisco, CA, June 2003.
8. C. Ding and K. Kennedy. Improving effective bandwidth through compiler enhancement of global cache reuse. *Journal of Parallel and Distributed Computing*, 64(1):108–134, 2004.
9. C. Fang, S. Carr, S. Onder, and Z. Wang. Instruction based memory distance analysis and its application to optimization. In *Proceedings of the International Conference on Parallel Architecture and Compilation Techniques*, St. Louis, MO, 2005.
10. G. Marin and J. Mellor-Crummey. Scalable cross-architecture predictions of memory hierarchy response for scientific applications. In *Proceedings of the Symposium of the Las Alamos Computer Science Institute*, Sante Fe, New Mexico, 2005.
11. R. L. Mattson, J. Gecsei, D. Slutz, and I. L. Traiger. Evaluation techniques for storage hierarchies. *IBM System Journal*, 9(2):78–117, 1970.
12. E. Petrank and D. Rawitz. The hardness of cache conscious data placement. In *Proceedings of ACM Symposium on Principles of Programming Languages*, Portland, Oregon, January 2002.
13. 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*, pages 381–391, San Diego, California, USA, June 2007.
14. D. D. Sleator and R. E. Tarjan. Amortized efficiency of list update and paging rules. *Communications of the ACM*, 28(2), 1985.
15. 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*, Charlottesville, Virginia, September 2002.
16. Y. Zhong, S. G. Dropsho, X. Shen, A. Studer, and C. Ding. Miss rate prediction across program inputs and cache configurations. *IEEE Transactions on Computers*, 56(3), 2007.