Using the Compiler to Improve Cache Replacement Decisions

Zhenlin Wang† Kathryn S. McKinley§ Arnold L. Rosenberg† Charles C. Weems†

† Department of Computer Science
University of Massachusetts, Amherst

§ Department of Computer Science
University of Texas, Austin

ABSTRACT

Memory performance is increasingly determining microprocessor performance and technology trends are exacerbating this problem. Most architectures use set-associative caches with LRU replacement policies to combine fast access with relatively low miss rates. To improve replacement decisions in set-associative caches, we develop a new set of compiler algorithms that predict which data will and will not be reused and provide these hints to the architecture. We prove that the hints either match or improve hit rates over LRU. We describe a practical one-bit cache-line tag implementation of our algorithm, called *evict-me*. On a cache replacement, the architecture will replace a line for which the evict-me bit is set, or if none is set, it will use the LRU bits. We implement our compiler analysis and its output in the Scale compiler. On a variety of scientific programs, using the evict-me algorithm in both the level 1 and 2 caches improves simulated cycle times by up to 34% over the LRU policy by increasing hit rates. In addition, a combination of simple hardware prefetching and evict-me works together to further improve performance.

1. Introduction

Microprocessor speeds have been steadily improving by about 55% per year since 1987. Meanwhile, memory access latencies have been improving only by 7% per year [12]. Cache hierarchies attempt to bridge this gap, and researchers have studied them intensely since their invention. To attain fast (1 or 2 cycles) cache access times, current microarchitectures have direct-mapped or low, 2 or 4-way, set-associative organizations [12, 13]. This choice trades off lower cache hit rates for higher clock rates to achieve better total performance. In set-associative caches, the architecture typically chooses to evict the least-recently-used (LRU) line on a replacement. LRU uses history, assuming that it should keep the most recently accessed data in the cache and evict the least recently accessed data. This organization and replacement policy does not always use cache memory effectively; i.e., even though the cache has sufficient capacity to retain data that will be reused in the future, LRU does not retain it [3, 6, 23]. This paper describes novel compiler and architecture mechanisms that use compiler prediction of future accesses to improve cache replacement decisions directly.

Consider the simple example in Figure 1. Notice that array B is accessed in nest 1 but not in nest 2. Whenever there is a cache miss in the first nest, we prefer to evict an element of array B. However, LRU ranks items from least to most recently used, i.e., A, B, C. Assuming that the cache size is a little bigger than 2*N, LRU will evict part of A even in a fully associative cache. A better replacement algorithm keeps both A and C and reuses them in nest 2. An optimal replacement algorithm however must know all future accesses, and hence is clearly infeasible [5, 32].

```
SUBROUTINE TEST(N)
INTEGER A(N), B(N), C(N)
   DO N1 I = 1,N
         C(I) = A(I)+B(I)
   ENDDO

   DO N2 I = 1:N
         A(I) = C(I) * 5
   ENDDO
END
```

Figure 1: A simple example

In this paper, we develop a new compiler mechanism that guides cache replacements by selectively predicting when data will or will not be reused. We develop a comparative model that uses dependence and array section analysis to determine static locality patterns in a program. We first prove that our model matches or improves hit rates when compared to LRU. We then present an implementation that uses a single tag bit called the *evict-me* bit. On a miss, the architecture replaces a line with this bit set. For example in Figure 1, we mark the evict-me bit for B on its load, and then evict it on a replacement to its cache set.

Our compiler algorithm aggressively marks data as evict-me if the data volume accessed between its reuse is (or it predicts the reuse is) greater than twice the cache size. The compiler can mark data aggressively, since if all the data fits in the cache, there will be no replacements. By applying the evict-me bit to both level 1 and level 2 caches, we observe up to 21% simulated performance improvements for current technology on a selection of scientific benchmarks and 34% for a technology prediction for 5 years from now [3]. On average, we reduce simulated execution time from 4.89% to 15.62% depending on the cache configuration.

Prefetching should be complementary to evict-me, since when effective, it fetches data that will be used in the future. We report results from a preliminary exploration of the interaction of evict-me with a simple hardware prefetching mechanism. We find that evict-me usually outperforms simple prefetching. In combination, evict-me helps alleviate cache pollution introduced by hardware prefetching and further improves cache performance. This paper thus introduces a new mechanism that enhances cache replacement decisions directly and we automate its use in a compiler.

This paper is organized as follows. Section 2 discusses related work. Section 3 presents potential hardware implementations. Section 4 introduces locality analysis and a new concept, *reuse level*. It also describes techniques for generating reuse levels at compile time. Section 5 presents an algorithm based on reuse levels and proves that it is at least as good as LRU and has the potential to

improve overall hit rates. We then describe an algorithm that uses a 1-bit tag for each cache line. Section 6 and Section 7 discuss our experimental methods and simulation results. Section 8 concludes the paper.

2. Related Work and Motivation

This section briefly discusses related work in set-associative cache design, prefetching, compiler algorithms for improving locality, and trace driven cache replacement algorithms.

Direct-mapped first level caches have been popular because of their low hit cycle time. They can yield good system performance, even though set-associative caches have lower miss rates [12, 13]. Due to rapid increases in miss cycle penalties, many recent architectures use at least 2-way set-associative first-level caches, e.g., the Compaq Alpha 21364 and Sun Sparc2. To attain single cycle level-one cache access in future technologies, processors will probably have small level one caches with a low degree of associativity [3]. Some architectures trade higher associativity with a simple cache replacement policy. For example, IBM RS/6000 7043 has 64K 128-way level 1 cache which uses random replacement policy. The hardware mechanisms of an evict-me cache do not increase cycle time and are only effective on set-associative caches; i.e., the hit time is unchanged. The replacement logic on a miss considers one more bit. Our work tries to achieve the hit rate of higher associativity by improving the replacement decision of a cache with lower associativity, thus achieving both fast hit cycle time and low miss rates.

The evict-me bit is similar to, but not the same as the Alpha's *evict* instruction which evicts a cache line immediately and thus cannot tolerate imprecision [17]. It is designed to maintain cache coherence, rather than enhance locality. Our approach works for variable cache and data sizes because only when the data does not all fit in the cache will the replacement algorithm use our information. The Alpha's *prefetch and evict-next* instruction loads the line to the level 1 cache and evicts it on the next miss to the cache set [17], but we instead tag actual loads, not speculative prefetches.

Our work takes an opposite approach as compared to hardware and software data prefetching which tolerate latency [4, 15, 19, 24, 26]. Data prefetching tries to fetch data which will be used in the near future to reduce miss penalties. Evict-me tags instead predict which data will and will not be used in the near future, and keep the data in the cache that will be used. Our technique eliminates misses, using the cache more effectively as compared to prefetching. Evict-me tags do not bring new data into the cache and thus do not have the higher bandwidth and other overhead of prefetching. Prefetching often pollutes caches when it brings in useless data. Evict-me tags can help alleviate the side effects of hardware prefetching. In Section 6, our preliminary results show that the combination of evict-me tags and hardware prefetching can further improve performance.

McKee et al. [21] propose a stream buffer to bypass stream-like data. We mark stream data as *evict-me*. But our technique works on cache replacement directly and does not require an extra buffer. The Intel IA-64 provides instructions to control caching [8]. The non-temporal load/store bypasses the cache to avoid cache pollution due to streaming data. IA-64 supports locality hints used by prefetch, load, and store instructions to control placements of cache lines in either a "temporal structure" or "non-temporal structure". The hints do not direct cache replacement, but our compiler analysis could specify the non-temporal instructions and locality hints. We do not explore this application here.

Numerous dynamic or hardware techniques have been proposed to reduce cache misses, e.g., [2, 14, 15]. The victim cache was originally designed to enhance direct-mapped caches [15]. It is a small fully-associative buffer between the level 1 and 2 cache which stores replaced data to reduce conflict misses that occur close together in time. It is probabilistic, rather than predictive. The evict-me bit works directly on a cache and replacement decisions. Johnson et al. [14] propose a run time spatial locality detection mechanism. They use a hardware table to keep track of spatial locality dynamically. The fetch size can be varied depending on the spatial locality of fetched data. Their work does not address cache replacement. Wong and Baer [35] enhance LRU with a temporal bit for each cache line. Temporal bits act oppositely to our evict-me bits: they specify lines to retain rather than lines to evict. Wong and Baer determine temporal bit settings using profiling or an online hardware history table. Rivers et al. [29] use a (hardware) detection unit, similar to a history table, to track reuses at run time and to categorize access as temporal/non-temporal and cacheable/non-cacheable. Lai et al. [18] use a hardware history table to predict when a cache block is dead and which block to prefetch to replace the dead one. Our technique is based on static compiler analysis and does not require substantially additional hardware.

Previous work studies the limits of cache performance using program traces. Belady [5] pioneered this area by comparing random cache replacement, LRU, and a new optimal algorithm. Sugumar and Abraham [31] used Belady's algorithm to characterize capacity and conflict misses. Temam [32] extended Belady's optimality result by simultaneously exploiting spatial and temporal locality. These studies seek to understand cache characteristics rather than to implement a real cache and related algorithms. Although our theoretical model in Section 5.1 is also based on static traces, we apply it to a real cache using compiler analysis. Ghosh et al. [9] suggest a set of miss equations for precisely analyzing cache misses for individual nests. This model could probably be extended to suggest evictions, but currently drives optimizations by comparing the number of misses between compiler options. Our work is less precise for an individual nest, but computes or estimates data volume between nests and between reuses. A better cache miss analysis could improve our results.

Researchers have also proposed loop and data transformations to improve data locality by moving temporal reuse closer together in time and by introducing spatial locality [1, 16, 22]. These algorithms do not directly improve replacement decisions and thus are complementary to our work.

On-line page coloring and other mechanisms decrease paging, but are too expensive for higher levels of the memory hierarchy. For example, Early Eviction LRU (EELRU) [30] dynamically chooses to evict the LRU page or the e^{th} most recently used page. Reference history determines e, the *early eviction point*, but is too expensive to store and use in caches. This approach eliminates capacity page misses in a fully associative memory, whereas our technique removes conflict misses for caches, using static compiler control.

3. Hardware Implementation

A simple implementation in the ISA is to duplicate a new set of memory instructions which set the evict-me tags and are otherwise the same as the the original set. We believe that the widening performance gap between memory and processor speeds must even-

tually be reflected by additional instructions in the ISA that help compensate for this gap. Hence, adding a new set of load and store instructions to the ISA is one step in this direction, and a simple step. However, our 1-bit evict-me replacement functionality can also be implemented without changes to the ISA in some architectures. For instance, on the Alpha 21264, we can first use the "prefetch and evict-next" instruction to set the evict-me bit and then perform a register load or store [17]. This implementation needs two loads or stores to set an evict-me bit and suffers inefficiency.

We use five extra bits in each memory instruction that the compiler sets to resolve run time spatial locality (see Section 5.2). An alternative hardware implementation uses a new instruction to store the 5-bit constant into a special register. The following memory operations will then access the special register and constant to detect spatial reuse. The compiler could use loop unrolling to avoid any extra instructions.

4. Locality

In this section, we briefly review locality, cache organizations that exploit it, and ideal replacement algorithms. We introduce our reuse notation and then present a new compiler algorithm that predicts locality within a loop nest (*intra-nest*) and between loop nests (*inter-nest*).

4.1 Perfect Locality Information: Trace-based Replacement

The reason caches perform well is that most programs exhibit good locality. The classical notions of locality found in programs are: *temporal locality* - if an item is referenced, it will be referenced again soon; and *spatial locality* - if an item is referenced, an adjacent item will tend to be referenced soon [12]. LRU takes advantage of program locality. It tries to keep the recently referenced data in cache and expects that data will be referenced again soon. However, as we pointed out earlier with regard to Figure 1, although arrays A, B and C all have spatial locality in nest 1, only arrays A and C have temporal locality due to reuses in nest 2. LRU can not exploit this fact because its decision is based on history. But the compiler can detect this information.

In our work, we want to approximate the locality of references in a given program. Consider the following quantitative definition of temporal locality [27]. The *temporal locality* of a data reference at time T is $TL = 1/(T_{next} - T)$, where T_{next} is the time of the next access to that particular address. We can similarly define spatial locality as follows. The *spatial locality* of a data reference at time T is $SL = 1/(T_{next} - T)$, where T_{next} is the time of the next access to the same cache block.

In this work, we assume the minimum unit of communication between main memory and the cache is a block: whenever any part of a block causes a miss, the architecture loads the entire block. Thus, in our model, temporal locality is a special case of spatial locality. If we know the temporal and spatial locality of each data reference in a program trace, then the optimal replacement algorithm replaces the data which has reuse furthest in the future, i.e., the data with the smallest value for SL [5]. Of course, computing TL and SL requires a complete trace which is not available at run time and is impossible to know exactly via static program analysis. To control cache replacement explicitly, we need a new method to describe locality. In the following section, we introduce *reuse level*, which is a measure that is comparative rather than absolute. We then show how to compute reuse levels using dependences.

4.2 Reuse Levels

Assume that we have a complete *trace* of a program: a series of references in the program following the execution order, i.e., $b_{f(1)}$, $b_{f(2)}$,..., $b_{f(n)}$. The subscripts are the block addresses which determine the references. The block addresses of the references need not be distinct, of course. *Reuse level* is used to approximate the locality of each reference. Rather than describe a specific distance from the current reference to the next reference to the same block, reuse levels describe a range in which the next reference will occur. Formally, the locality of reference $b_{f(i)}, 1 \leq i \leq n$, is a set $s \in \mathcal{S}_n$, where
$\mathcal{S}_n = \{[n+1, +\infty]\} \cup \{[j,k] \mid 1 \leq j \leq k \leq n\}$ *and* $[j,k] = \{j, j+1, .., k\}$.

1. If s is $[n+1, +\infty]$, then block $b_{f(i)}$ will not be referenced again after the i_{th} reference; i.e., $f(i) \neq f(l)$ for all $l, i < l <= n$.

2. If s is $[j,k]$ for some j, k, $i < j \leq k \leq n$, then $\exists t, j \leq t \leq k$, such that the next reference to block $b_{f(i)}$ is the t_{th} reference in the trace, i.e., $f(i) = f(t)$, and $f(t) \neq f(l)$ for all $l, i < l < t$.

   Then we call the set s the *reuse level* of $b_{f(i)}$. To compare reuse levels for references, we define three relations on $\mathcal{S}_n$:$\prec$, $\sim$, and $\succ$.

$[i,j]$	$\prec$	$[n+1, +\infty]$	for all	$1 \leq i \leq j \leq n$
$[i,j]$	$\prec$	$[k,l]$	if	$j < k$
$[i,j]$	$\sim$	$[k,l]$	if	$[i,j] \cap [k,l] \neq \phi$
$[i,j]$	$\succ$	$[k,l]$	if	$[k,l] \prec [i,j]$

Theorem 1. Only one relation holds for any two elements in $\mathcal{S}_n$.

Proof. By definition. □

Theorem 1 shows that reuse levels are comparable. Intuitively, if two blocks conflict, we want to replace the block whose reuse level is $\prec$ than that of the other block. When two reuse levels are $\sim$ to each other, we use access history to break ties (as does LRU).

4.3 Using Dependences as Reuse Levels

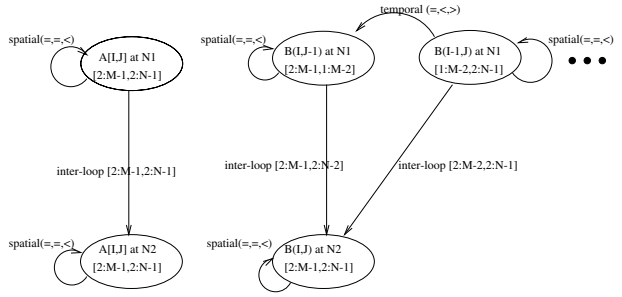
This section explains how to combine dependences with the loop iteration space to produce reuse levels. We assume the reader is familiar with dependence analysis which detects reuse between array references in a loop nest [10, 28]. We use dependence testing to detect temporal and spatial reuses within a loop nest. To detect reuses between distinct loop nests, we use bounded regular sections [11] to describe the access range of a reference in a loop nest. The descriptors for bounded regular sections (BRSD) are vectors of elements, each of which is a triplet. A triplet describes an accessing range in a dimension, consisting of a lower bound, an upper bound, and a step. The bounded regular section for A(I,J) in Figure 2(a) is $[2 : M-1 : 1, 2 : N-1 : 1]$. The descriptors support union and intersection operations. There is a reuse between two references in distinct nests if the intersection set of their BRSDs is not empty.

We build a *locality graph* based on reuses. The graph describes temporal and spatial locality within each loop nest and across loop nests. An edge connecting two references in the same loop nest contains the reuse vector. An edge connecting two references in distinct loops contains the intersection of the two BRSDs. Figure 2(b) shows the locality graph for the sample program in Figure 2(a), where for simplicity we omit B(I+1,J) and B(I,J+1). In

```
PROGRAM SimplifiedJacobi
    PARAMETER (N=1000, M=1000)
    REAL A(N, M), B(N, M)

    DO J = 2 , N-1
        DO I = 2, M-1
            A(I, J) = (B(I-1, J)+B(I+1, J)+B(I, J-1)+B(I,J+1))/ 4
        ENDDO
    ENDDO
    DO J = 2, N-1
        DO I = 2, M-1
            B(I, J) = A(I, J)
        ENDDO
    ENDDO
END
```

(a) Another simple program



(b) Locality graph

Figure 2: A simple program and its locality graph

Step/order		0	1	2	3	4	5	6
PREDICTION	block 1		r1 $< [3,4], 1 >$	r1 $< [3,4], 1 >$	r1 $< [3,4], 1 >$	r1 $< [21,28], 4 >$	r2 $< [10,12], 5 >$	r2 $< [10,12], 5 >$
	block 2			r2 $< [5,6], 2 >$	r3 $< [5,6], 3 >$	r3 $< [5,6], 3 >$	r3 $< [5,6], 3 >$	r3 $< [10,12], 6 >$
	miss/hit		miss	miss	miss	hit	miss	hit
LRU	block 1		r1	r1	r3	r3	r2	r2
	block 2			r2	r2	r1	r1	r3
	miss/hit		miss	miss	miss	miss	miss	miss

Table 1: LRU versus Prediction for a 2-way set-associative cache

Figure 2(b), the first element of a reuse vector denotes the inter-nest reuse direction. If it is '=', the reuse is in the same nest. If it is '<'. then the dependence is inter-nest. Now the vector $(=,<,>)$ from B(I-1,J) to B(I,J-1) denotes an intra-nest input dependence and a temporal reuse across J loop.

We can rely on reuse vectors as predictors of access patterns. We can use those vectors as reuse levels if we also add information that describes the relative position between independent references. We can either track the loop iterations at run time or keep the reuse levels up to date as different instances of a reference execute. Now a reuse level is a set of loop iteration points which consist of run time memory references. For example, we can use the direction vectors shown in Figure 2(b) as reuse levels with the following semantics. The B(I,J-1) has a self spatial reuse with direction vector $(=,=,<)$. The direction vector by itself means there is a spatial reuse due to a later reference to B(I,J-1) itself in the same nest, the same J iteration, but the later I iterations. As a reuse level, the direction vector means the iteration points from the next I iteration through I=M-1. Specifically, given an loop iteration at I=5 and J=4, the run time instance of reference B(I,J-1) is B(5,3), whose reuse level $(=,=,<)$ means it has a reuse between iteration I=6 and iteration I=M-1 under J=4. To illustrate our idea, let's take off the spatial reuse vector of B(I-1,J). Now Reference B(I-1,J) only has a temporal reuse with vector $(=,<,>)$ which means the reuse is in the later J iterations. It is obvious that $(=,=,<) \prec (=,<,>)$. So when B(5,3) and B(4,4) conflict, our cache replacement policy will choose the cache line of B(4,4) to evict. We now describe cache replacement algorithms that use reuse levels.

5. Cache Replacement Algorithms

In this section, we show how to improve cache replacement decisions in an ideal case and within the context of realistic cache organizations. First, we develop a general framework that is guaranteed to match or improve hit rates over LRU given sufficient hardware support. We then present a simple, but practical one-bit encoding, called the *evict-me* bit, that indicates when a cache block is a good choice for replacement.

5.1 Improving LRU Cache Replacement

Our first cache replacement algorithm, the Prediction algorithm, uses the access order of a reference and its reuse level to direct replacement. Consider a program trace $b_{f(1)}^{<s_1,1>}, b_{f(2)}^{<s_2,2>}, ..., b_{f(n)}^{<s_n,n>}$, where $b_{f(i)}$ is the i_{th} block accessed by address $f(i)$, and $< s_i, i >$ are its reuse level and access order respectively. We define a relation $\triangleleft$ on the set $\mathcal{Q}_n = \{< s_i, i >, s_i \in \mathcal{S}_n, 1 \leq i \leq n\}$, as follows:
$< s_i, i > \triangleleft < s_j, j >$ if $(s_i \prec s_j)$ or $(s_i \sim s_j$ and $i > j)$.

Each <*reuse level, order*> pair is an element of $\mathcal{Q}_n$.

Theorem 2. For each pair of elements in $\mathcal{Q}_n$, $< s_i, i >$ and $< s_j, j >$, $i \neq j$, either $< s_i, i > \triangleleft < s_j, j >$ or $< s_j, j > \triangleleft < s_i, i >$.

Proof. By definition.□

The Prediction algorithm updates a reference's order and its reuse level in the cache on every access. Think of a cache set as an ordered list from smallest to largest by the $\triangleleft$ ordering of the <reuse level, order> pairs. Initially every reuse level is the $[n+1, \infty]$, and on a reference, the architecture sets the reuse level if it is specified. Whenever there is a miss, the last line with the largest <reuse level, order> pair is replaced. When a reference changes the cache line's <reuse level, order> pair, we change its position in the list. We compare it to the other items in the list from first to last until the $\triangleleft$ ordering of the line is smaller than that of the next element, and then insert the line before this next element. Although the $\triangleleft$ ordering is not a partial order (because it is not transitive), the definition of the Prediction algorithm and the list ordering algorithm guarantees that there is a deterministic ordering of the list after each cache access; i.e., Theorem 1 and 2 are sufficient to ensure that the

Prediction algorithm is totally specified.

The following example illustrates the algorithm. Assume a two-way set associative cache and a simple program trace $a_{r1}^{<[3,4],1>}$, $a_{r2}^{<[5,6],2>}$, $a_{r3}^{<[5,6],3>}$, $a_{r1}^{<[21,28],4>}$, $a_{r2}^{<[10,12],5>}$, $a_{r3}^{<[10,12],6>}$, ..., all of whose elements are mapped into a single cache set. Here r1, r2, and r3 are references to distinct blocks in main memory. The content of the cache is shown in Table 1. In step 3, LRU replaces r1, which leads to a miss in step 4. However, since $< [3,4], 1 > \lhd < [5,6], 2 >$, the Prediction algorithm replaces r2 instead. In this example, it performs better than LRU, which results in all misses.

Theorem 3. For the same cache configuration (same cache size, same degree of associativity, and same block size), at each reference point, if there is an LRU hit, there is also a Prediction hit.

Proof. See appendix.□

Theorem 3 tells us that the Prediction algorithm is at least as good as the LRU algorithm at any reference point. So if we can find a reuse level for each reference point, we expect to improve upon the LRU algorithm. In Section 4.3, we have shown that dependence vectors combined with loop iterations can predict reuse distances.

5.2 Evict-me: 1-Bit Encoding

The Prediction algorithm can be implemented by encoding reuse levels into memory instructions. We previously studied a 16-bit encoding [34] which serves as a useful upper bound on the compiler accuracy. Using 16 auxiliary bits for each cache line will increase the time to determine which line to replace and may consume too much area. For an 8K level one cache with 32-byte cache line, 16 extra bits would contribute about 5% to the cache area.

There are two ways to address these problems. One is to implement the policy in lower-level caches where the cost of extra reuse level bits and the comparison latency are relatively low. For example, a 256K level two cache with 128-byte cache line only need devote 1.5% additional area to annotations. The other way is to simplify the model. A 16-bit encoding implies up to 2^{16} reuse levels. The evict-me tag denotes two reuse levels, s_1 (no reuse) and s_0 (reuse). We combine it with the LRU bits as we discussed in Section 5.1. The Prediction algorithm performs as following. If the evict-me bit of a block is set, the replacement algorithm will choose that block to replace on a miss. Otherwise, it follows the LRU policy. The compiler generates special-purpose instructions to set evict-me bits and thus explicitly control cache replacement.

This one-bit encoding suggests that we classify reuse distances into two levels such that a distance vector in one level is always less than one in the other level. A simple and very conservative algorithm tags the array references which have no locality in a loop nest and are not reused in any following nests. Assume the total number of run time memory accesses in a routine of a nest is n. In the nest, the algorithm uses two reuse levels, $s_0 = [1, n]$ for references with reuses on this nest or subsequent nests, and $s_1 = [n+1, +\infty]$ for references with no reuses in this subroutine. Following the definition in Section 4, we have $[1, n] \prec [n+1, +\infty]$. A more aggressive algorithm follows Theorem 4.

Theorem 4. In a w-way set-associative cache, if the number of distinct references mapped into the same set between a reference and its reuse is greater than w, then evicting the first reference in the next replacement will not degrade the overall LRU hit rate.

```
setEvictMeTag()
{
   for each loop nest {
      compute nest volume
      for each array reference r in the nest {
         if (r has no temporal reuse in this nest) {
            if (nest volume > 2 * cache size)
               mark r evict-me
            else if (volume unknown && nest level ≥ 2)
               mark r evict-me
            else if (r has no temporal reuse with the next nest)
               mark r evict-me
            if (r has spatial reuse) set reference step
         }
      }
   }
}
```

Figure 3: Algorithm for setting evict-me tag

Proof. See appendix.□

We can design an algorithm which sets the EM tag when accessing a reference without reuse or with reuse that is sufficiently far away. Accurately counting the number of distinct references mapped to a specific cache set is impossible at compile time when the iteration counts and size of arrays are unknown. We estimate these sizes at compile time. If the loop bounds of the nest are all constants and available at compile time, we combine them with the BRSDs to compute the exact data volume. When the loop bounds of a nest are unknown, we use a simple heuristic which assumes that the data volume of a nest is greater than two times the level one cache size if it contains more than one level of loop nesting.

Now, if we can determine the total data volume between a reference and its reuse across loop nests, and it is greater than twice the cache size, then we predict it will not be reused; i.e., that the number of distinct references mapped into a set between the two reference will be greater than the degree of associativity. This intuition implies that Theorem 4 holds.

Figure 3 presents the aggressive algorithm for singling out references without temporal or spatial reuse in a nest. It never marks references with temporal intra-nest reuse. It sets the evict-me bit for those references whose reuse spans more than two times the cache size, or when the data volume is unknown, whose nesting depth is 2 or more, or if the reference has no temporal reuse with the adjacent nest. If the reference has spatial locality on any loop level, the compiler marks it, such that the architecture will exploit it before marking it for eviction. In the program in Figure 2(a), we set the evict-me tags of A(I,J) in both nests, the tag of B(I,J-1) in nest 1, and that of B(I,J) in nest 2, because these four references have no temporal reuses and the total data volume of each nest (near 8 $*10^6$) is greater than twice the cache size. We are able to mark very aggressively because the evict-me bit is only examined on a miss, when the architecture needs to replace something.

In our implementation, we encode the spatial locality information of a reference into the memory instruction and let the hardware detect it at run time. Formally, we consider only arrays with the least significant index in the form of $a * I + b$, where I is the loop induction variable, and a and b are constants. We assume that the

	Conf. 1	Conf. 2	Conf. 3
Level 1	8K, 2-way	32K, 2-way	64K, 4-way
	32 byte cache line		
Level 2	128K, 2-way	256K, 4-way	512K, 2-way
	128 byte cache line		

Table 2: Three cache configurations

loop step of the induction variable I is a constant s. Let p be the word position of the reference in the cache line, l be the cache line size, e be the element size in the number of words, and $a * s$ be the *reference step*. If $a * s$ is positive, the reference has self-spatial reuse when $p < l - a * s * e$. If $p > -a * s * e$ and $a * s$ is negative, the reference also has self-spatial reuse. Similar techniques resolve group-spatial reuse. The reference step is usually very small. We use five bits to encode the reference step. For a reference with evict-me tag marked by the compiler, if it also has spatial locality, the run time environment waits to set the bit until after the spatial reuse is complete.

5.3 Effectiveness of Evict-me Algorithm

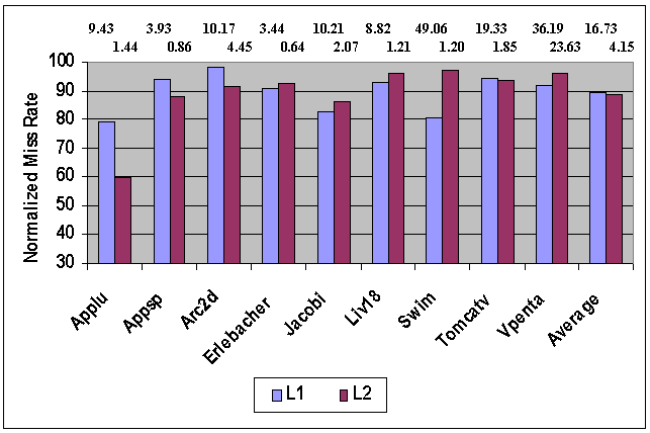
The evict-me algorithm is sensitive to both program access patterns and cache configurations. For a specific program with a specific input, evict-me bits can be very effective in one cache configuration but help little in the other. Take the simple program in Figure 1 as example. Assume that N is 4K, a word size is 4 bytes, and the starting address of array A is aligned to 16K. In a 32K 2-way level 1 cache, A(I), B(I), and C(I) will be map to the same set for each I. In this case, B(I) annotated with the evict-me bit will help reduce inter-nest misses. However, in a 64K 2-way level 1 cache, array A and B will map to different cache areas and thus evict-me will perform exactly the same as LRU. Given a complicated application which contains many loops and different access patterns, evict-me will yield a better cache replacement for some and not other inputs and cache configurations.

6. Experimental Results

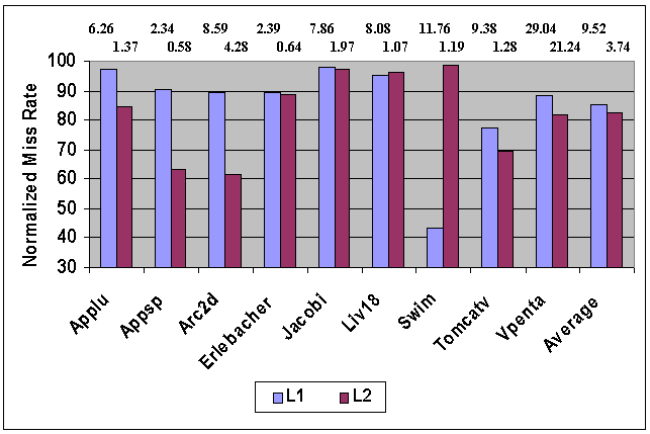
6.1 Compiler, Simulators, and Benchmarks

For our experiments, we use Scale, a compiler infrastructure developed by our research group. It accepts C and Fortran source code as input. The system translates the source code into an intermediate representation called Scribble. Scribble keeps the high-level program structures such as loop and array references and, at the same time, the low-level operations. Our dependence testing is based on the Omega library [28, 33]; we used its algorithms and interfaces, but rewrote it in Java, since the rest of Scale is in Java. We implemented the analyses described in Section 4 and Section 5 and performed them on Scribble. We apply on Scribble a set of optimizations such as sparse conditional constant propagation, loop interchange, global value numbering, partial redundancy elimination, and scalar replacement. We then output Sparc assembly. The evict-me tag and reference step of a memory instruction are encoded into an unimplemented Sparc instruction. We put the marking instruction before the memory instruction. We use the level 1 cache size to drive our algorithm. The simulator we use preprocesses the machine code and merges the two instructions.

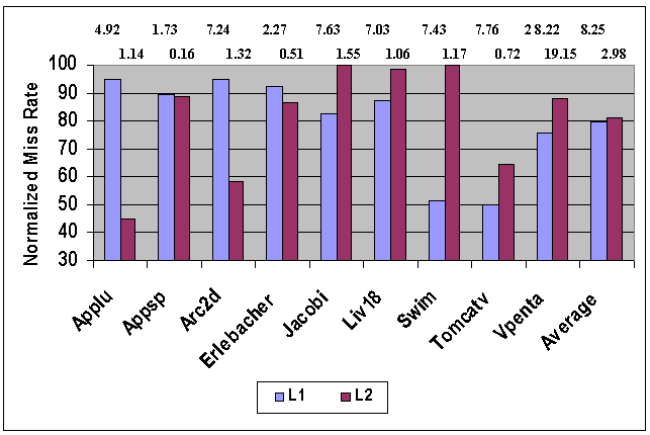
We use URSIM developed at the University of Utah to simulate the performance impact of the evict-me cache [36]. URSIM is an extension to RSIM, which simulates a state-of-the-art out-of-order processor, lock-up free cache, and multi-bank memory [25]. Scale generates Sparc assembly code with annotated load/store instructions for URSIM. We updated URSIM to accept the special



(a) Conf. 1



(b) Conf. 2



(c) Conf. 3

Figure 4: Miss reduction by Evict-me

load/store instructions and perform the corresponding replacements.

We use nine benchmarks. Liv18, Vpenta, Erlebacher, and Jacobi are kernels. Swim, Tomcatv, and Applu are from Spec95. Arc2d is a Perfect benchmark and Appsp is from the Nas Benchmarks. We selected benchmarks that had high miss rates or loop nest structures with inter-nest misses, and that ran through our compiler.

We configure URSIM to model a 4-way issue and out-of-order processor core. We use two levels of cache which are all non-blocking, and have eight miss status handler registers (MSHRs) each. Evict-me can be turned on/off in both levels of the cache. We apply three level 1 and level 2 cache combinations of sizes and associativities, as shown in Table 2. The three configurations share the same cache line size and latencies. The level 1 cache line size is 32 bytes and the latency 2 cycles. The level 2 cache line size is 128 bytes and latency 8 cycles. The latency for memory access is between 48 and 200 cycles and depending on the state of the machine; this range reflects the sophistication of the accurate memory model. We also examine all our benchmarks using a 5 year hardware projection where the level 2 latency is increased to 20 cycles and the memory access latency is 200-500 cycles. These projections come from Agarwal et al.[3].

6.2 Miss Rates Results

Figure 4(a) through Figure 4(c) show the normalized miss rates at the three cache configurations when the evict-me replacement is turned on for both level 1 and level 2 caches. At the top of each bar, the LRU miss rate is listed. For example, at configuration 1, the miss rate of Applu by LRU at level 1 is 9.43%. Evict-me reduces the miss rate by 21%. The miss rate of level 2 is the level 2 misses with respect to total accesses rather the misses of the level 1 cache. We use this representation to show the combined effects of evict-me on the two levels of cache. The miss reduction in the level 2 cache comes not only from better replacements in the level 2 cache itself, but also from better level 1 replacements which reduce the traffic between the two level caches.

We observe a significant miss reduction for both levels of cache. As we discussed in Section 5.3, evict-me could be very effective in one cache configuration but less so in others. For Applu, the miss rate at level 1 is reduced by 21.13% in configuration 1 but only 1.37% in configuration 2. At certain cache configurations, we reduce the miss rate of Applu, Swim, and Tomcatv by about 50%. Overall, the miss reduction ranges on average from 10% to 20%. Evict-me never degrades the miss rate, although mispredictions resulting from our heuristics in Section 5 might cause a degradation.

6.3 Static and Dynamic Replacement Counts

Table 3 shows static and dynamic statistics on evict-me tags and their effect on replacements for our programs. The second column is the percent of annotated instructions among all static load and store instructions. We mark 25% of the memory instructions on average at compile time. The numbers in the remaining columns are collected under the cache configurations of a 64K L1 4-way cache and a 512K L2 2-way cache (Conf. 3), with evict-me caching on in both caches. The third and the fifth columns are the percent of cache accesses in which we set the evict-me bit in the level 1 and 2 caches respectively. The fourth and the sixth columns show the percent of replacements where the evict-me bit changes the replacement decision as compared to LRU's decision. It changes 4% to 24% of the decisions in the L1, and 0 to 15% in the L2. These changes do not correspond well to changes in miss rates because one change can result in several more hits, or no additional hits. For example, evict-me removes many misses for Tomcatv and Swim but alters 13% or fewer L1 replacement decisions. The changes to level 2 replacements are on average very low which means that the L2 miss reductions also come from better L1 replacements and more L1 hits that yield less traffic between the caches.

	Static	Dynamic (Conf. 3)			
	evict-me	L1 evict-me	L1 Repl.	L2 evict-me	L2 Repl.
Applu	12.05	13.81	6.66	30.41	3.44
Appsp	8.93	9.06	15.04	24.46	9.61
Arc2d	25.96	20.61	4.29	36.30	14.86
Erlebacher	25.36	12.64	12.62	15.68	13.16
Jacobi	50.00	33.95	24.10	56.36	2.35
Liv18	43.82	30.01	21.61	53.31	1.42
Swim	20.95	21.51	8.30	53.35	0.21
Tomcatv	9.52	4.89	13.03	4.98	0.57
Vpenta	35.05	15.69	11.71	25.57	3.51
Average	25.74	18.02	13.04	33.38	5.46

Table 3: Static and dynamic statistics on evict-me

6.4 Simulated Performance Results

Table 4 shows the performance impact of evict-me. The columns titled "L1" and "L2" show performance improvements when the evict-me caching is turned on for the level 1 cache only and for level 2 cache only respectively. The columns titled with "L1+L2" are the improvements when the evict-me caching is turned on for both caches. For current technology, we see reductions in execution time of 4.89%, 8.39%, and 6.94% on average for the three configurations. We see larger improvements in simulation cycle time, 9.99%, 15.62%, and 12.31%, for predicted technology for 5 years from now when the gap between processor speed and memory speed increases. Usually and on average, the performance improves most when the evict-me caching is turned on in both caches. We see more contribution from the level 2 cache in most cases because the gap between the access time of the level 2 cache and memory is relatively larger than the gap between the two caches. In our experience, out-of-order execution often hides L1 cache latencies, but not L2 [20].

An interesting case is Vpenta which improves the most with evict-me turned on only in the level 2 cache at configuration 1 and 2. When evict-me is on in both caches, the level 1 evict-me cache replacements change the access pattern of the level 2 cache, and in this case, reduce its effectiveness. In Swim, the opposite is true; the level 1 cache dominates the evict-me performance improvements because the level 1 miss rate is very high and evict-me reduces it by 19% - 56%.

6.5 A Less Aggressive Compiler Marking Algorithm

We also investigate a slightly more conservative compiler algorithm for setting the evict-me bit. We use the same algorithm from Figure 3, except we change the test for nest level ≥ 2, to be ≥ 3, and thus mark fewer references as evict-me. With this algorithm, our results are unchanged for Jacobi, Liv18, and Vpenta because the compiler computes the data volume precisely. For Applu, Appsp, and Arc2d, this more conservative algorithm is slightly better, but for Tomcatv and Swim, it does not set enough bits, and the original is much better.

6.6 Combination of Evict-me and Hardware Prefetching

Hardware prefetching has been widely studied to hide cache latencies [4, 15, 19, 24, 26]. Prefetched cache blocks may pollute the cache if the blocks are useless. Several techniques seek to re-

	8K L1, 128K L2 (Conf. 1)				32K L1, 128K L2 (Conf. 2)				64K L1, 512K L2 (Conf. 3)			
	Current			5 years out	Current			5 years out	Current			5 years out
Program	L1	L2	**L1+L2**	**L1+L2**	L1	L2	**L1+L2**	**L1+L2**	L1	L2	**L1+L2**	**L1+L2**
Applu	10.58	0.37	**11.30**	**31.91**	0.24	3.92	**4.17**	**34.21**	10.08	10.04	**11.74**	**25.86**
Appsp	0.32	2.57	**2.60**	**8.50**	0.27	5.22	**4.93**	**16.65**	0.19	0.55	**0.49**	**2.53**
Arc2d	0.00	4.89	**4.81**	**7.93**	0.00	21.97	**21.59**	**30.22**	0.00	9.87	**9.48**	**26.03**
Erlebacher	0.48	1.10	**1.20**	**3.27**	0.66	1.10	**1.46**	**4.67**	0.37	1.37	**1.48**	**5.92**
Jacobi	2.05	4.85	**5.17**	**11.88**	0.00	5.19	**0.79**	**1.89**	1.57	0.00	**1.40**	**2.18**
Liv18	0.32	1.17	**1.50**	**2.26**	1.35	0.64	**2.17**	**2.67**	2.12	0.48	**2.54**	**2.78**
Swim	10.57	1.48	**9.48**	**11.30**	11.46	1.72	**11.23**	**11.68**	6.59	0.00	**6.53**	**6.38**
Tomcatv	0.66	1.98	**2.45**	**7.01**	5.72	3.23	**7.30**	**13.60**	7.66	0.00	**7.62**	**16.05**
Vpenta	0.00	11.03	**5.51**	**5.83**	0.31	32.00	**21.91**	**24.98**	0.21	20.03	**21.21**	**23.07**
Average	2.78	3.27	**4.89**	**9.99**	2.22	8.33	**8.39**	**15.62**	3.20	4.70	**6.94**	**12.31**

Table 4: Percent performance improvement by evict-me

	8K L1, 128K L2 (Conf. 1)			32K L1, 128K L2 (Conf. 2)			64K L1, 512K L2 (Conf. 3)		
	Evict-me	Prefetching	Evict-me+ Prefetching	Evict-me	Prefetching	Evict-me+ Prefetching	Evict-me	Prefetching	Evict-me+ Prefetching
Applu	11.30	-0.36	0.16	4.17	1.02	13.34	11.74	-0.59	13.44
Appsp	2.60	0.89	3.03	4.93	0.87	5.92	0.49	0.95	1.50
Arc2d	4.81	-8.79	-3.13	21.59	-7.79	16.84	9.48	0.48	11.74
Erlebacher	1.20	-1.97	2.53	1.46	-0.70	2.94	1.48	0.06	3.27
Jacobi	5.17	3.62	9.83	0.79	2.37	6.85	1.40	5.96	3.03
Liv18	1.50	3.01	4.43	2.17	2.99	4.52	2.54	2.88	4.22
Swim	9.48	1.62	9.17	11.23	2.02	14.95	6.53	1.01	8.60
Tomcatv	2.45	-0.17	2.33	7.30	-0.21	7.07	7.62	1.13	9.49
Vpenta	5.51	-13.24	-3.01	21.91	-10.80	14.36	21.21	-9.64	6.81
Average	4.89	-1.71	2.82	8.39	-1.14	9.64	6.94	0.25	6.90

Table 5: Combining Evict-me and Prefetching

duce cache pollution. For instance, hardware can detect strided accesses and selectively prefetch blocks which are expected to be useful [26]. Evict-me can help reduce the impact of cache pollution introduced by hardware prefetching in two ways. First, the cache pollution by a prefetched cache block may be harmless if it evicts a block which is marked as evict-me. In this situation, the marked line is probably useless anyway. Second, compilers can use locality analysis to decide when prefetching is necessary. We can mark a prefetched block as evict-me if our confidence on its locality is low. We explore the first option by combining a simple level 1 hardware prefetching mechanism implemented in URSIM with evict-me. The current level 1 cache prefetching in URSIM prefetches the next adjacent cache block when there is a cache miss and the level 1 request queue is not full. The hardware drops the prefetching request if there is no MSHR available. Table 5 shows the performance impact of the level 1 prefetching. We notice that this simple prefetching mechanism frequently degrades performance. However, the combination of prefetching and evict-me outperforms either of them in most cases at cache sizes of 32K (Conf. 2) and 64K (Conf. 3). For Applu at 32K, prefetching and evict-me improve execution times by 1.02% and 4.17% respectively. However, the combination introduces a performance improvement of 13.34%.

Figure 5 shows the normalized cycles at configuration 3 for LRU, evict-me, level 1 prefetching and the combination of evict-me and prefetching. We compare them with the cycles under perfect caches. The cycles under perfect L1 are based on the assumption that all level 1 accesses are hits. We apply the same assumption to collect cycles for perfect level 2 cache where the level 1 cache still uses the LRU replacements. When there is a performance gap of about 20% or more between LRU and perfect L1, such as in Applu, Arc2d, Swim, Tomcatv, and Vpenta, either evict-me or its combination with prefetching shows significant performance improvement. An interesting case is Swim where evict-me beats perfect L2. This

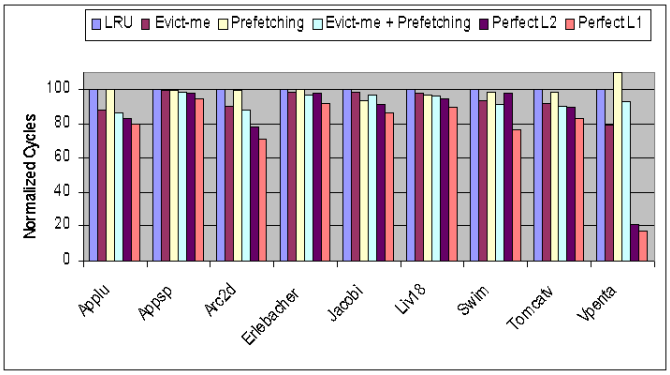


Figure 5: Evict-me and Prefetching versus Perfect Caches (Conf. 3)

result is because of its dominating level 1 miss rate and 48% miss reduction by evict-me on the level 1 cache.

7. Other Comparisons

We also did some miss rate explorations using SimpleScalar [7]. These results are not directly comparable to those we report above because they use very different ISAs and architectures. Above we use URSIM on Sparc binaries. With SimpleScalar, we use C code generated by Scale. We then compile it with gcc, and generate PISA binaries. PISA is an ISA specific to the simulator. Scale performs different optimizations and produces very different code than gcc. However, the miss rates in both studies follow the same trends. In experiments with SimpleScalar, we also compared evict-me to an optimal replacement algorithm, to higher set-associativity, and to the victim cache [34]. The victim cache and evict-me sometimes worked well on different programs, i.e., neither subsumes the other.

They always worked better together than on their own. Sometimes they were better than optimal replacement due to good use of the small additional space provided by the victim cache.

8. Conclusions

In this paper, we develop a theoretical model for static compilation analysis to direct cache replacement algorithms and prove that it is at least as good as LRU. This work opens a new path for reducing cache misses by compiler hints to improve replacement decisions. We present and implement a 1-bit evict-me version of our algorithm. We demonstrate that the 1-bit evict-me algorithm is practical enough to implement in current set-associative caches, and in multiple levels of the cache hierarchy. Furthermore, our simulation results show that the evict-me algorithm consistently improves performance through reduced miss rates when compared with LRU and is very effective on multiple levels of the cache. We combine the evict-me algorithm with hardware prefetching. We find that evict-me can negate the effects of cache pollution introduced by prefetching and further improve performance.

9. Acknowledgments

Thanks to David Culler for his insights on architectural trends. We also thank Lixin Zhang who developed much of the URSIM simulator. This work is supported by NSF ITR grant CCR-0085792, NSF grant ACI-9982028, NSF grant EIA-9726401, Darpa grant 30602-98-1-01015, Darpa grant F33615-01-C-1892, and IBM. Any opinions, findings and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect those of the sponsors.

10. REFERENCES

[1] W. A. Abu-Sufah. *Improving the Performance of Virtual Memory Computers*. PhD thesis, Dept. of Computer Science, University of Illinois at Urbana-Champaign, 1978.

[2] A. Agarwal and S. D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct-mapped caches. In *Proceedings of the 20th International Symposium on Computer Architecture*, pages 169–178, San Diego, CA, May 1993.

[3] V. Agarwal, M. Hrishikesh, S. W. Keckler, and D. Burger. Clock rate versus IPC: The end of the road for conventional microarchitectures. In *Proceedings of the 27th International Symposium on Computer Architecture*, pages 248–259, June 2000.

[4] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing '91*, Albuquerque, NM, Nov. 1991.

[5] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):79–101, 1966.

[6] D. Burger, A. Kägi, and J. R. Goodman. Memory bandwidth limitations of future microprocessors. In *Proceedings of the 23rd International Symposium on Computer Architecture*, Philadelphia, PA, May 1996.

[7] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.

[8] C. Dulong. The IA-64 architecture at work. *IEEE Computer*, pages 24–32, July 1998.

[9] S. Ghosh, M. Martonosi, and S. Malik. Precise miss analysis for program transformations with caches of arbitrary associativity. In *Proceedings of the Eighth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 228–239, San Jose, CA, Oct. 1998.

[10] G. Goff, K. Kennedy, and C. Tseng. Practical dependence testing. In *Proceedings of the SIGPLAN '91 Conference on Programming Language Design and Implementation*, pages 15–29, Toronto, Canada, June 1991.

[11] P. Havlak and K. Kennedy. An implementation of interprocedural bounded regular section analysis. *IEEE Transactions on Parallel and Distributed Systems*, 2(3):350–360, July 1991.

[12] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1995.

[13] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, Dec. 1988.

[14] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time spatial locality dection and optimization. In *Proceedings of the 30th International Symposium on Microarchitecture*, Research Triangle Park, NC, Dec. 1997.

[15] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully-associative cache and prefetch buffers. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 364–373, Seattle, WA, June 1990.

[16] M. Kandemir, A. Choudhary, J. Ramanujam, and P. Banerjee. A matrix-based approach to the global locality optimization problem. In *The 1998 International Conference on Parallel Architectures and Compilation Techniques*, Paris, France, Oct. 1998.

[17] R. E. Kessler, E. McLellan, and D. Webb. The Alpha 21264 microprocessor architecture. Technical report, http://www.compaq.com/AlphaServer/download/ev6chip.pdf, Nov. 1999.

[18] A. Lai, C. Fide, and B. Falsafi. Dead-block prediction & dead-block correlating prefetchers. In *Proceedings of the 28th International Symposium on Computer Architecture*, June 2001.

[19] W. Lin, S. K. Reinhardt, and D. Burger. Reducing dram latencies with an integrated memory hierarchy design. In *Seventh International Symposium on High Performance Computer Architecture*, pages 301–312, Monterrey, Mexico, Jan. 2001.

[20] G. Lindenmaier, K. S. McKinley, and O. Temam. Load scheduling with profile information. In A. Bode, T. Ludwig, and R. Wismüller, editors, *Euro-Par 2000 – Parallel Processing*, volume 1900 of *Lecture Notes in Computer Science*, pages 223–233. Springer-Verlag, Aug. 2000.

[21] S. A. McKee, R. H. Klenke, K. L. Wright, W. A. Wulf, M. H. Salinas, J. H. Aylor, and A. P. Batson. Smarter memory: Improving bandwidth for streamed references. *IEEE Computer*, pages 54–63, July 1998.

[22] K. S. McKinley, S. Carr, and C. Tseng. Improving data locality with loop transformations. *ACM Transactions on Programming Languages and Systems*, 18(4):424–453, July 1996.

[23] K. S. McKinley and O. Temam. Quantifying loop nest locality using SPEC'95 and the Perfect benchmarks. *ACM Transactions on Computer Systems*, 17(4):288–336, Nov. 1999.

[24] T. Mowry, M. S. Lam, and A. Gupta. Design and evaluation of a compiler algorithm for prefetching. In *Proceedings of the Fifth International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 62–73, Boston, MA, Oct. 1992.

[25] V. S. Pai, P. Ranganathan, and S. V. Adve. RSIM reference manual (version 1.0). Technical Report Technical Report 9705, Rice University, Dept. of Electrical and Computer Engineering, Aug. 1997.

[26] S. Palacharla and R. E. Kessler. Evaluating stream buffers as a secondary cache replacement. In *Proceedings of the 21th International Symposium on Computer Architecture*, pages 24–33, Chicago, IL, Apr. 1994.

[27] M. Prvulovi, D. Marinov, Z. Dimitrijevic, and V. Milutinovic. A survey annd reevaluation of performance. *IEEE TCCA Newsletters*, pages 8–17, 1999.

[28] W. Pugh. A practical algorithm for exact array dependence analysis. *Communications of the ACM*, 35(8):102–114, Aug. 1992.

[29] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *Proceedings of the 1997 ACM International Conference on Supercomputing*, pages 449–456, Melbourne, Australia, July 1998.

[30] Y. Smaragdakis, S. Kaplan, and P. Wilson. Eelru: Simple and effective adaptive page replacement. In *Proceedings of the ACM SIGMETRICS Conference on Measurement & Modeling Computer Systems*, pages 122–133, Atlanta, GA, May 1999.

[31] 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*, pages 24–35, Santa Clara, CA, May 1993.

[32] O. Temam. An algorithm for optimally exploiting spatial and temporal locality in upper memory levels. *IEEE Transactions on Computers*, 48(2):150–158, Feb. 1999.

[33] Univerisity of Maryland. *The Omega Library*, 1996. http://www.cs.umd.edu/projects/omega/.

[34] Z. Wang, K. S. McKinley, and A. L. Rosenberg. Improving replacement decisions in set-associative caches. In *Proceedings of MASPLAS'01, The Mid-Atlantic Student Workshop on Programming Languages and Systems*, Hawthorne, NY, Apr. 2001.

[35] W. A. Wong and J. Baer. Modified lru policies for improving second-level cache bahavior. In *Sixth International Symposium on High Performance Computer Architecture*, pages 49–60, Toulouse, France, Jan. 2000.

[36] L. Zhang. URSIM reference manual. Technical Report UUCS-00-015, University of Utah, Aug. 2000. http://www.cs.utah.edu/projects/impulse.

APPENDIX

Proof of Theorem 3. The proof is based on the trace we defined at the beginning of Section 5.1.

Say that we are working on a w-way set associative cache. Assume, for contradiction, that at reference $b_{f(i)}^{<s_i,i>}$ there is a miss for Prediction algorithm and a hit for LRU. Let $b_{f(j)}^{<s_j,j>}$ be the nearest reference to the same block address where $j < i$. We know that $f(i) = f(j)$.

Claim 1: There are no more than w distinct references mapped into the same set between $b_{f(j)}^{<s_j,j>}$ and $b_{f(i)}^{<s_i,i>}$ included.

To simplify the discussion, assume that each block in a set is aged from 1 to w by the access order. The block with the smallest order has age w, the one with the largest order has age 1. With the LRU algorithm, the block with age w is evicted when there is a miss. At the time when $b_{f(j)}^{<s_j,j>}$ is brought into the cache, its age is 1. Assume for contradiction that at least w distinct references are different from $b_{f(j)}$ between $b_{f(j)}^{<s_j,j>}$ and $b_{f(i)}^{<s_i,i>}$ are mapped into the same set. All those w references have greater order than that of $b_{f(j)}$, so each reference will increase the age of $b_{f(j)}$ by 1. Thus, when we access the $w-1_{th}$ reference of that kind, $b_{f(j)}$ has age w. The access to the w_{th} reference of that kind will evict $b_{f(j)}$, and no reference will bring $b_{f(j)}$ back because $b_{f(i)}^{<s_i,i>}$ is the most recent reference to block $b_{f(j)}$. This contradicts the hit at $b_{f(i)}^{<s_i,i>}$.

Next assume that there is an age between 1 and w associated with each block in the list defined for Prediction cache in Section 5.1. The ages of the blocks are consistent with the ordering of the list. The $< reuse\ level, order >$ pair of the block at age 1 is smaller in $\lhd$ ordering than the pair of the next block in the list, and so on. We let t_i denote the time when the i_{th} reference gets accessed and assume after time Δt, the access completes. We have $t_i + \Delta t < t_{i+1}$ for all i.

Say now that $b_{f(j)}$ has age m at time $t_j + \Delta t$. Because we have a miss of $b_{f(i)}^{<s_i,i>}$ at time t_i, there exists a reference $b_{f(k)}^{<s_k,k>}$ at time t_k, for some $j < k < i$, which is also a miss and $b_{f(j)}$ has age w when the reference $b_{f(k-1)}^{<s_{k-1},k-1>}$ completes.

Claim 2: All addresses in the cache set at time $t_{k-1} + \Delta t$ are referenced at least once between time t_j and t_i.

Let $r_{j,1}, r_{j,2}, ..., r_{j,m-1}, r_{j,m}, .., r_{j,w}$ be the block addresses in the cache set at time $t_j + \Delta t$ and $r_{k,1}, r_{k,2}, ..., r_{k,w}$ be those at time $t_{k-1} + \Delta t$, where the second subscript denotes the age of the corresponding address. We know that $r_{j,m} = r_{k,w} = f(j)$. Let $U = \{r_{j,1}, r_{j,2}, ..., r_{j,m-1}\} \cap \{r_{k,1}, r_{k,2}, ..., ., r_{k,w}\}$.

First, all addresses in $S = \{r_{k,1}, r_{k,2}, ..., ., r_{k,w}\} - U$ must be referenced between time t_j and $t_{k-1} + \Delta t$. Assume, for contradiction, that $r_l \in S$ is not referenced during this period, then $r_l \in S$ must be referenced before time t_j, and must be in the cache set at time t_j since it is in the cache set at time $t_{k-1} + \Delta t$. Then since r_l is not in $\{r_{j,1}, r_{j,2}, ..., r_{j,m-1}\}$ because it is in S, it has an older age than that of $b_{f(j)}$ at time $t_j + \Delta t$. No reference can change this relationship unless the two references themselves are accessed again between time $t_j + \Delta t$ and t_{k-1}. Note that $b_{f(j)}$ has age w at time $t_{k-1} + \Delta t$. r_l must be evicted before time t_{k-1} since it is older. Then it can not be in the set at time $t_{k-1} + \Delta t$, contrary to assumption.

Second, all references in U must be referenced between time t_j and t_i. Notice that all references in U have $< reuse\ level, order >$ pairs $\lhd$ than that of $b_{f(j)}$ at time $t_j + \Delta t$ just after $b_{f(j)}$ is brought into cache. Since the orders of these references are less than j, by the definition of relation $\lhd$, they must have smaller reuse levels which means they will be referenced before the next reference to $b_{f(j)}$, which occurs at time t_i.

The references in the w-block set at time $t_{k-1} + \Delta t$ are distinct. Furthermore, the reference $b_{f(k)}$ is distinct from the blocks in the set since it is a miss. The total number of distinct references mapped into the set between time t_j and t_i are at least $w + 1$. Contradiction. □

Proof of Theorem 4. Let's say we are working on a w-way set associative cache and a program trace $\mathcal{P}$. We focus on a specific cache set $\mathcal{C}$. Assume that sub-trace $b_{f(1)}^{s_1}, b_{f(2)}^{s_2}, ..., b_{f(n)}^{s_n}$ is the largest subset of $\mathcal{P}$ mapped into set $\mathcal{C}$ in its original order. $b_{f(i)}$ is the i_{th} block mapped into $\mathcal{C}$, and its block address is $f(i)$. s_i is the evict-me tag going with the access. In particular, $s_i = 0$ means it is a regular access; $s_i = 1$ means the block's evict-me tag gets set after this access. Following the condition of the theorem, we have the following assertion,

Assertion: If $s_i = 1$, then $j - i > w$ for any access $b_{f(j)}$ where $f(j) = f(i)$ and $j > i$.

Now we prove that for any access $b_{f(j)}^{s_j}$ in the sub-trace, if LRU results in a hit, then there is a hit for evict-me at this access. Assume that we get an LRU hit at $b_{f(j)}^{s_j}$. Let access $b_{f(i)}^{s_i}$ be the closest reference to block $b_{f(j)}$ where $i < j$. Since it is an LRU hit, we have $j - i <= w$ (we proved this in the proof of Theorem 3). Now $s_i = 0$ follows the assertion. Since $s_i = 0$, the evict-me algorithm can at most increase the age of $b_{f(i)}$ by 1 at each following reference. So at access $b_{f(j-1)}^{s_{j-1}}$, the age of $b_{f(i)}$ should be less than w. Evict-me algorithm leads to a hit at $b_{f(j)}^{s_j}$. □