

Improving Replacement Decisions in Set-Associative Caches

Zhenlin Wang Kathryn S. McKinley Arnold L. Rosenberg

Computer Science Department, University of Massachusetts, Amherst

ABSTRACT

Cache replacement policies play a key role in determining hit rates in set-associative caches. Cache replacement algorithms use runtime trace history and most a least recently used (LRU) policy, and neither programmers nor compilers can explicitly control cache replacement. This paper describes a novel mechanism to improve cache replacement decisions without the hardware costs of higher set-associativity. We develop compiler-generated auxiliary information using dependence testing and locality analysis to improve cache replacement decisions for scientific programs. We present an ideal theoretical model of our new cache replacement algorithm and prove that it matches or improves LRU for set-associative caches. We present a 16 and 1-bit cache line tag implementation of this model. We call the one bit tag the *evict-me* (EM) bit. On a miss, the architecture replaces a line if its EM bit is set, or, if no bit is set, it uses the LRU bits. We prove that the EM bit yields replacement decisions at least as good as those of LRU. The EM bit is practical and easy to implement in current set-associative cache architectures. Simulation results on 7 programs show that the EM bit can reduce miss rates in set-associative caches by up to 45% over LRU. On 6 of the 7 programs, it achieves the same or better hit rates with 2-way set associative as compared to a 4-way cache without an EM bit. A comparison with victim cache shows that the two strategies can work together to further improve cache 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 [10]. Cache hierarchies attempt to bridge this gap, and researchers have studied them intensely since their invention. To attain fast cache access times, current microarchitectures have direct-mapped or low, 2 or 4-way, set-associative organizations [11, 10]. In set-associative caches, the architecture chooses to evict the least-recently-used (LRU) line on a replacement. An optimal replacement algorithm must peek into the future, and hence is clearly infeasible [6, 22]. LRU uses past 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, the LRU policy does not [3, 7, 17].

Consider the simple example in Figure 1. Notice that array B is accessed in nest N1 but not in nest N2. 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., C, B, A. Assuming that the cache size is a little bigger than $2 \cdot N$, LRU will evict part of A even in a fully associative cache. A better replacement algorithm can keep both A and C until their reuse in nest 2. In this paper, we develop a new compiler mechanism that guides cache replacements by predicting when data will be reused. We use dependence and array section analysis to determine static locality patterns in a program. The compiler then encodes when data will be reused. On a replacement, the architecture chooses to evict data that the compiler predicts will be reused furthest in the future.

We first prove that our model matches or improves hit rates in a fully associative cache when compared with LRU. We then develop algorithms for encoding reuse in cache-line tag bits for set-associative caches, and prove that these algorithms will similarly match or improve hit rates compared with the LRU policy. In particular, we show how to use a single tag bit called the *evict-me* (EM) bit. On a miss, the architecture will choose to replace a line with this bit set. For example in Figure 1, we mark the EM bit for B on its load, and then evict it on a replacement to its cache set. The EM bit was inspired by the Alpha's *evict* instruction which evicts the cache line immediately, and the *prefetch and evict-next* instruction which loads the line to the level 1 cache and evicts it on the next miss to the cache set [15]. These instructions have similar, but different semantics as compared to the EM bit. Our results show that the EM bit can improve miss rates by up to 45% on a 4-way set associate cache, and on 6 of 7 of our benchmarks enables a 2-way set associative cache with EM bits to improve or match a normal 4-way cache. Our approach thus attains the hit rate of higher set-associativity without the hardware costs on our benchmarks. Results for a 16-bit tag show further improvements are possible. We also implement a victim cache for level 1 cache. The results demonstrate that neither strategy dominates the other. However, using both strategies further improves cache performance.

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 provides a proof that it is at least as good as LRU and has potential

```

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

to in fact improve the overall hit rates. We then describe algorithms that use 16 and 1-bit tags for each cache line. Section 6 discusses our experimental methods and simulation results. Section 7 concludes the paper.

2. Related Work and Motivation

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

Direct-mapped first-level caches have been popular because their lower hit cycle time, as compared to set-associative caches, often yielded better system performance, even though set-associative caches have lower miss rates [11, 10]. However due to the rapid increases in miss penalties in cycles, many recent architectures use at least 2-way set associative first level caches, e.g., the Compaq Alpha 21364 and Sun Sparc2. In current technologies, the hit cycle time penalties of even higher degrees of associativity still negate their reduced miss rates. To attain single cycle level one cache access in future technologies, processors probably have small level one caches with low degree of associativities [3]. The hardware mechanisms of an EM cache do not increase cycle time and are only effective on set associative caches; i.e., the hit time is unchanged although 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 low hit cycle time and low miss rates.

Our work takes an opposite approach as compared with hardware and software data prefetching which tolerate latency [13, 5, 18, 19]. Data prefetching tries to fetch data which will be used in the near future to reduce miss penalties. EM 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 techniques eliminate misses, using the cache more effectively as compared to prefetching. EM tags do not bring new data into the cache and thus do not have the bandwidth and other overheads of prefetching. However, prefetching does not require a set-associative cache.

A number of dynamic or hardware techniques have been proposed to reduce cache misses [2, 12, 13]. The victim cache was originally designed to enhance direct-mapped caches by keeping a small number of replaced lines in a fully-associative buffer [13]. It also improves set-associative caches. We can apply our techniques to the victim buffer to improve its replacement decisions. We compare an EM cache with a cache augmented with a victim cache with and without EM bits in Section 6.

Previous work proposes a run-time spatial locality detection mechanism [12]. It uses a hardware table to keep track of spatial locality dynamically. The fetch size can be varied depending on the spatial

locality of fetched data. The IBM Power3 and the Cray T3E implement aggressive hardware prefetching in the second level cache, but neither addresses cache replacement.

Previous work uses program traces to study the limits of cache performance. Belady pioneered this area by comparing random cache replacement, LRU, and optimal algorithms [6]. Sugmar and Abraham used Belady’s algorithm to accurately characterize the notion of capacity and conflict misses [21]. Recently, Temam extended Belady’s optimality result by simultaneously exploiting spatial and temporal locality [22]. All 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 by using compiler analysis.

Another research trend is to improve cache locality with compiler techniques. Abu-Sufah first discussed applying loop transformations to improve paging [1]. McKinley et al. used a cost model to capture more types of reuses [16]. All this work focuses on single loop nest. Kandemir et al. improves cache performance for a sequence of loop nests through a combination of loop and data layout transformations [14]. We believe loop transformations will make locality prediction more accurate, and can exploit more fully the performance of our replacement strategy. We plan to study these interactions in the future.

3. Hardware Implementation

We believe that the widening performance gap between memory and processor speeds must eventually be reflected by additional instructions in the ISA that help compensate for this gap. Hence, adding new load and store instructions to the ISA that set EM tags is one step in this direction, and a simple step. However, our 1-bit EM replacement functionality can also be implemented without changes to the ISA in some architectures. On the Alpha 21264 which has a 2-way set associative level one cache, we can first use the “prefetch and evict-next” instruction to set the EM bit and then perform a register load or store [15]. The “prefetch and evict-next” instruction loads the data into the level 1 cache, and then replaces it on the next miss to the same set. Other implementations that do not change the ISA are also possible. For example, we could directly use the physical addresses of the level 1 cache to manipulate the EM bit.

4. Locality

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

4.1 Background

To simplify further discussion, we concentrate on a set-associative, write-allocate, write-back cache. 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. We use miss rates to compare the performance of different cache replacement algorithms.

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 [10]. LRU takes advan-

```

Nest 1: DO  $i_{11} = l_{11} : u_{11} : s_{11}$ 
    ...
    DO  $i_{1n} = l_{1n} : u_{1n} : s_{1n}$ 
        body 1
    ENDDO
    ...
ENDDO
...
Nest k: DO  $i_{k1} = l_{k1} : u_{k1} : s_{k1}$ 
    ...
    DO  $i_{kn} = l_{kn} : u_{kn} : s_{kn}$ 
        body k
    ENDDO
    ...
ENDDO

```

Figure 2: An abstract program

tage of program locality. It tries to keep the data recently referenced 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 are reused in nest 2. LRU can not exploit this fact because its decision is based on history. In this paper, we explore a new mechanism for using locality information to improve cache replacement decisions.

4.2 Perfect Locality Information: Trace-based Replacement

In our work, we want to approximate the locality of references in a given program. Consider the following quantitative definition of temporal locality [20]. 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 consecutive access to that particular address. Since we focus on caches that bring a whole block of data in at a time, 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 consecutive access to the same 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 [6]. Of course, we do not have a complete trace at program execution, and SL 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 a measure which is comparative rather than absolute.

4.3 Compiler Approximated Locality Information

This section first reviews dependence analysis. Then we abstract dependence distance by a new representation, called *reuse level*. We show that reuse levels are comparable and thus can be used as a basis for a cache replacement algorithm.

Consider the abstract program shown in Figure 2: all nests in the program contain the same number of loops. The *iteration space* of loop nest i is an n -dimensional polyhedron consisting of all the n -tuples of values of the loop indices, called an *index vector*. Extending the idea of iteration space, we define the iteration space of the whole program as all valid values of vector $\langle j, i_{j1}, i_{j2}, \dots, i_{jn} \rangle$, called a *inter-loop index vector*. Here the value of j , $1 \leq j \leq k$, denotes the subspace of the j_{th} loop nest. We use \prec to denote the lexicographic ordering of inter-loop index vectors.

$\langle j, i_{j1}, i_{j2}, \dots, i_{jn} \rangle \prec \langle j', i'_{j1}, i'_{j2}, \dots, i'_{jn} \rangle$ if $j < j'$ or ($j = j'$ and for some $l, \forall 1 \leq m \leq l, i_{jm} = i'_{jm}$ and $i_{jl} < i'_{jl}$)

There is a *dependence* between two references in a program if there exist two runtime instances of the references that refer to the same memory address. In fact, dependence denotes temporal reuse of a reference, i.e., multiple accesses to the same memory location. Dependence testing determines if two references to the same array access the same memory location by testing if there is a solution to a set of linear equations which makes the subscripts equal to each other. Dependence testing detects spatial reuse when the equations make all subscripts equal except the least significant one, and the difference between the two least significant subscripts is less than the cache block size. Testing returns a *distance* or *dependence vector* which describes the distance, or a range of distances, for which the two index vectors satisfy the dependence equations. When specific distances cannot be obtained through static compilation analysis, compilers use direction vectors to describe the ordering of the source and target index vectors. In our implementation, we have both, but the encodings we present in Section 5 use direction vectors since they are more general.

To detect dependences between distinct loop nests, we use bounded regular sections [9] 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 3(a) is $[2 : M - 1, 2 : N - 1]$. The default step is 1. The descriptors support union and intersection operations. There is a dependence between two references in distinct nests if the intersection set of their BRSDs is not empty. We build a *locality graph* based on dependences. The graph describes temporal and spatial reuses within each loop nest and across loop nests. It is a directed graph consisting of nodes that correspond to the references in the program, and edges that describe the locality of related nodes. An edge connecting two references in the same loop nest contains the dependence vector. An edge connecting two references in distinct loops contains the intersection of the two BRSDs. Figure 3(b) is the locality graph for the sample program in Figure 3(a), where for simplicity we omit $B(I+1,J)$ and $B(I,J+1)$.

We can rely on the dependence distance as an oracle of access patterns. If we can track the loop iterations at runtime, then the compiler can tell at which iteration points reuse will occur. Notice that what the dependence distance predicts is not an exact point of reuse, but a range. We thus need a method to compare ranges.

We introduce a new representation called *reuse level* which has dependence distance as a special case. Assume that we have a complete *trace* of a program: a series of references in the program following the execution order, i.e., $a_{b_{f(1)}}^1, a_{b_{f(2)}}^2, \dots, a_{b_{f(n)}}^n$. The superscripts show the order of accesses, and the subscripts are the blocks with their addresses as subscripts 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 $a_{b_{f(i)}}^i, 1 \leq i \leq n$ is a set $s \in \mathcal{S}_n$, where

$$\mathcal{S}_n = \{\phi\} \cup \{s_{jk} | 1 \leq j \leq k \leq n\} \text{ and } s_{jk} = \{j, j+1, \dots, k\}.$$

1. If s is ϕ , then block $b_{f(i)}$ will not be referenced again after the i_{th} reference; i.e., $f(i) \neq f(l)$ for all $i < l \leq n$.
2. If s is s_{jk} for some $j, k, i < j \leq k \leq n$, then $\exists t, j \leq t \leq k$,

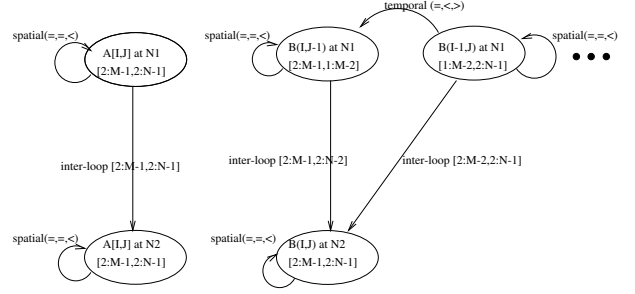
```

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 3: A simple program and its locality graph

	step	0	1	2	3	4	5	6
PREDICTION	cache block1		r1	r1	r1	r1	r2	r2
	cache block2			r2	r3	r3	r3	r3
	miss/hit		miss	miss	miss	hit	miss	hit
LRU	cache block1		r1	r1	r3	r3	r2	r2
	cache block2			r2	r2	r1	r1	r3
	miss/hit		miss	miss	miss	miss	miss	miss

Table 1: LRU versus Prediction

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

$$\begin{aligned}
s_{ij} &\prec \phi && \text{for all } 1 \leq i \leq j \\
s_{ij} &\prec s_{kl} && \text{if } j < k \\
s_{ij} &\sim s_{kl} && \text{if } s_{ij} \cap s_{kl} \neq \phi \\
s_{ij} &\succ s_{kl} && \text{if } s_{kl} \prec s_{ij}
\end{aligned}$$

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

Proof. By definition. \square

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).

4.4 Using Dependences as Reuse Levels

Reuse direction vectors can be directly used as reuse levels but have different semantics. If we consider the reuse direction vectors shown in Figure 3(b) as reuse levels, they can be compared with each other following the definition of reuse level. We explain the semantics of this special reuse level using the reference B(I,J-1) in Figure 3(a) as an example. The spatial reuse of B(I,J-1) has direction vector $(=, =, <)$. The direction vector by itself means there is a dependence between B(I,J-1) and a reference in the same nest, the same J iteration, but later I iterations. As a reuse level, the direction vector means the iteration points from the next I iteration until the iteration when I=M-1. For a particular runtime instance of reference B(I,J-1), say B(5,J-1), its reuse level $(=, =, <)$ means it has a reuse between iteration I=6 and iteration I=M-1. We are

now ready to describe cache replacement algorithms that use reuse levels generated by locality analysis at compile time.

5. Cache Replacement Algorithms

In this section, we show how to use locality information to improve cache reuse in an ideal case and within the context of realistic cache organizations. First, we develop an abstract algorithm that can improve hit rates over LRU. Then we use our model to develop an algorithm for improving replacement decisions in a k -way set associative cache, where k is assumed to be small (e.g., $2 \leq k \leq 4$, as in most modern microprocessors). The second algorithm uses 16 extra bits to encode reuse levels, which is impractical in current microarchitectures, but is useful for investigating the limits of our approach. We further explore an algorithm that uses only one bit of extra information called the *evict-me* bit that indicates when a cache block is a good choice for replacement. We prove that these algorithms will improve or match a k -way set-associative cache when compared to LRU.

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 $a_{b_{f(1)}}^{\langle s_{1,1}, 1 \rangle}, a_{b_{f(2)}}^{\langle s_{2,2}, 2 \rangle}, \dots, a_{b_{f(n)}}^{\langle s_n, n \rangle}$, where $b_{f(i)}$ is the i_{th} block accessed by address $f(i)$, and $\langle s_i, i \rangle$ are its reuse level and access order respectively. We define a relation \triangleleft on the set $\mathcal{Q}_n = \{ \langle s_{jk}, i \rangle, s_{jk} \in \mathcal{S}_n, 1 \leq i \leq n \}$, as follows:

$$\langle s_{jk}, i \rangle \triangleleft \langle s_{mn}, l \rangle \text{ if } (s_{jk} \prec s_{mn}) \text{ or } (s_{jk} \sim s_{mn} \text{ and } i > l).$$

Each $\langle \text{reuse level}, \text{order} \rangle$ pair is an element of \mathcal{Q}_n .

Theorem 2. For each pair of elements in \mathcal{Q}_n , $\langle s_{jk}, i \rangle$ and $\langle s_{mn}, l \rangle$, $i \neq l$, either $\langle s_{jk}, i \rangle \triangleleft \langle s_{mn}, l \rangle$ or $\langle s_{mn}, l \rangle \triangleleft \langle s_{jk}, i \rangle$.

$\langle \langle s_{jk}, i \rangle \rangle$.

Proof. By definition. \square

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 \langle ordering of the \langle reuse level, order \rangle pairs. Initially every reuse level is the ϕ , 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 \langle reuse level, order \rangle pair is replaced. When a reference changes the cache line’s \langle reuse level, order \rangle pair, we change its position in the list. We compare it to the other items in the list from first to last until the \langle ordering of the line is smaller than that of the next element, and then insert the line before this next element. Although the \langle 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 have a simple program trace $a_{r_1}^{\langle [3,4], 1 \rangle}$, $a_{r_2}^{\langle [5,6], 2 \rangle}$, $a_{r_3}^{\langle [5,6], 3 \rangle}$, $a_{r_1}^{\langle [21,28], 4 \rangle}$, $a_{r_2}^{\langle [10,12], 5 \rangle}$, $a_{r_3}^{\langle [10,12], 6 \rangle}$, ..., all of whose elements are mapped into a single set. Here $[i, j]$ is the set $\{i, i + 1, \dots, j\}$, and r_1 , r_2 , and r_3 are distinct block addresses. Then the content of the cache is shown in Table 1. In step 3, LRU replaces r_1 , which leads to a miss in step 4. However, since $\langle [3, 4], 1 \rangle < \langle [5, 6], 2 \rangle$, the Prediction algorithm replaces r_2 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 [23]. \square

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.

5.2 16-Bit Encoding

The Prediction algorithm makes replacement decisions based on reuse levels. In Section 4.4, we discussed how to obtain reuse levels for each array reference. To use reuse levels at runtime, we assume an extended ISA that has extra bits for setting the reuse levels in each cache line on loads and stores, and a cache that supports these bits. We discussed other implementation options in Section 3. Cache tag bits are set when a memory instruction with tag bits is executed. Cache replacement is based on the value of cache tag bits and LRU history bits. The 16-bit method we describe here is not practical for an implementation but lets us explore more fully the accuracy of our reuse information and encoding. We encode here the reuses in each loop nest and the reuses between two adjacent nests. For inter-nest reuse, we only consider adjacent loops because most inter-nest misses occur between adjacent two nests [17]. We use a 16-bit annotation because the simulator we use supports at most a 16-bit annotation. This encoding gives us a loose upper bound on our technique. Looking for a better encoding is left to future work. Figure 5 shows our algorithm for using these bits. The function of each bit is shown in Table 2. The encoding as-

```

...
L: DO i = l : u : s
N1: DO i1 = l1 : u1 : s1
    ... R ...
    ENDDO
N2: DO i2 = l2 : u2 : s2
    ... R ...
    ENDDO
ENDDO
...

```

Figure 4: A loop nest

Bit	Function
15	temporal bit for level 4
14	spatial bit for level 4
13	temporal bit for level 3 (inter-nest bit for level 4)
12	spatial bit for level 3
11	temporal bit for level 2 (inter-nest bit for level 3)
10	spatial bit for level 2
9	temporal bit for level 1 (inter-nest bit for level 2)
8	spatial bit for level 1
7	temporal bit for level 0 (inter-nest bit for level 1)
6	spatial bit for loop level 0
5	sign of reference step (1:negative, 0: positive)
4-1	reference step
0	reuse level tag

Table 2: Encoding for 16-bit reuse level

sumes that the deepest level of a loop nest is 4, which is appropriate for most applications. We assume for each routine there is a virtual loop enclosing the whole routine. The virtual loop is at level 0. For each level, we have a spatial bit and a temporal bit. The bit for a reference at loop L is set if the reference has reuse cross iterations or reuse in the current iteration. Bit 0 is set when the compiler can determine the reuse levels of a reference. The temporal bit of loop level i also functions as an inter-nest temporal reuse bit for a nest whose outermost loop is at level $i + 1$. There is no conflict for this bit to serve two functions. Consider loop L at level l whose loop body consists of two nests, $N1$ and $N2$, as shown in Figure 4. A reference R in $N1$ has an inter-nest reuse in $N2$. Its inter-nest bit for level $l + 1$ is set because the outermost loop of nest $N1$ is at level $l + 1$. The same bit also serves as the temporal bit for level l which means a reference has reuse in the the current or future L iterations. These semantics are the same as the semantics of the inter-nest bit which means the reuse is in the current iteration.

Figure 7 shows the program in Figure 3(a) with 16-bit reuse levels, which are listed in hexadecimal form. We use reference $B(I+1, J)_{\langle 0x0e83 \rangle}$ as an example to explain our algorithm. $B(I+1, J)$ has both spatial and temporal reuse at loop level 2 (the I loop), so the 10_{th} and 11_{th} bits of its reuse level are set to 1. $B(I+1, J)$ has temporal reuse at loop level 1 (the J loop), so the 9_{th} bit is set to 1. It also has inter-nest reuse and the outermost loop of the nest is at level 1, so the 7_{th} bit is set to 1. Bit 0 is set because compiler knows all reuses of the reference.

Note that in our encoding, we try to put all reuse levels of a reference together. A static reference usually has different reuse levels for different loops or nests. For instance, in Figure 3(a), $A(I, J)$ in nest 1 has spatial reuse across the I loop. It also has inter-nest temporal reuse. We need two reuse levels for the reference. At run time, we should always first use the smallest reuse level in the \rightarrow

ordering. In fact, this mechanism is implicitly shown in our encoding where we assign more significant bits to deeper loops.

Now the Prediction algorithm can make cache replacement decision simply based on the value of reuse levels associated with each cache line. It always evicts the cache line with the smallest reuse level. A special case is when the reuse level of a cache line is 0, the eviction is based on its access order as in LRU. Here our implementation is more aggressive than the formal definition of the Prediction algorithm in Section 4. In the implementation, we keep each cache set in its LRU ordering. When there is a miss in a set, the last line of the set is replaced if its reuse level is 0, which means that compiler does not know its reuse level. Otherwise, the algorithm chooses the line with the smallest reuse level.

The reuse level of a reference usually does not span the whole subspace of the reference particularly for spatial reuses. In the locality graph shown in Figure 3(b), given, for example, a cache line size of two words and $A(I,J)$ is aligned on the cache line for all J , we notice that the spatial reuse of $A(I,J)$ occurs only when I is odd. If we know the starting address of each array reference, loop unrolling can produce array references with and without spatial locality. Our implementation uses another method to resolve this problem. At run time, we know the cache block size and exactly where a memory block will be mapped into a cache line. This information and the access pattern of an array reference are usually enough to decide if there is spatial reuse. For example, for the $A(I,J)$ we just mentioned, we know the next access to array A is $A(I+1,J)$. Hence we can be sure the reference $A(I,J)$ will have no spatial reuse across the I loop if it is mapped into the last word of a cache line. For a given array reference whose indices are all affine expressions, the compiler discovers the pattern of spatial reuse and encodes it into the corresponding instruction. 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 also assume that the 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 encoded into reuse levels using bit 1 to bit 5. Obviously, in Figure 7, $B(I+1,J)$'s reference step is positive and its value is 1, so bit 1 to bit 5 are set to 1.

In our implementation, we insert an `update()` function as shown in Figure 6 at the exit of each loop. The function expires those reuse levels which are no longer valid. The `update()` function in Figure 6 can help to reduce the side effects of miss-prediction for both spatial and temporal reuse, because it expires the prediction of reuse in a loop after the execution of the loop. If only a small percentage of predictions are not correct, they will expire sooner or later, and will not affect the miss rate too much. In the example code, we notice that the temporal reuse of $B(I-1,J)$ exists for all J except $J=N-1$. The miss-predictions at $J=M-1$ are insignificant and the `update()` function between the two nests will expire them. For our encoding, the implementation of the `update(int l)` function needs to set the temporal and spatial bit at level l to 0. In Figure 7, note that after the I loop finishes its execution, `update(2)` is executed which sets bit 10 and bit 11 of all reuse levels to 0. It expires the prediction to the reuses across the iterations of loop I . Similarly, `update(1)` expires the prediction to the reuses across the iterations

```
reuseLevelGeneration()
{
  for each perfect nest whose outermost loop is at level  $j$  {
    for each array reference  $r$  in the nest {
      reuseLevel = 0;
      for each loop at level  $i$  enclosing the reference  $r$  {
        if  $r$  has temporal reuse across the loop iterations
          /* including the current iteration*/
          set the  $(6+2^i)$ th bit of its reuseLevel to 1
        if  $r$  has spatial reuse across the loop iterations
          /* including the current iteration*/
          set the  $(5+2^i)$ th bit of its reuseLevel to 1
          set the value of 1-5th bit by the reference step and its sign of  $r$ 
        if  $r$  has inter-nest reuse then
          set the  $(4+2^i)$ th bit of its reuseLevel to 1
      }
    }
  }
}
```

Figure 5: 16-bit Reuse-level Generation

```
update(int l)
{
  for all reuse levels associated with each cache block {
    remove those predicting reuses for level  $l$ 
  }
} /* end update */
```

Figure 6: Update Function

of loop J in the first nest. But the prediction to the inter-nest reuses is kept live.

5.3 Evict-Me: 1-Bit Encoding

Using 16 auxiliary bits for each cache line will increase the time to determine which line to replace and may consume too high percentage of cache space. A 8K level one cache with 32-byte cache line has to contribute about 5% of cache size to those annotations. The cost of the update function is also probably too high. There are two ways to address these problems. One is to implement it 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% of total cache to annotations. The other way is to simplify the model and apply it in high-level caches. Here we use a one-bit (EM) *evict-me* tag. If the EM bit of a block is set, the replacement algorithm will choose that block to replace a miss. Otherwise, we follow the standard LRU policy. For a w -way set-associative cache, a naive method needs $\log w$ bits to keep LRU history as a pointer for each cache line. LRU evicts the cache line with the largest pointer value. With the EM policy, the EM bit is added to each pointer as the most significant bit and the replacement logic remains the same. The compiler generates special-purpose instructions to set the EM bit and thus explicitly control cache replacement. Theorem 3 tells us if the tagged blocks in a set are always reused later than the blocks without tags, we can at least match the hit rate of LRU. Intuitively, we want to tag the data that has poorest locality.

A simple heuristic algorithm is to tag the array references which are not reused in the following nests. Assume the total number of references in a program is t . The algorithm sets up two reuse levels, $[1, t]$ and $[t + 1, +\infty]$. Following the definition in Section 4, we have $[1, t] \prec [t + 1, +\infty]$. 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

```

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

DO J = 2, N-1
DO I = 2, M-1
A(I, J)<0.X0483> = (B(I-1, J)<0.X0683> +
B(I+1, J)<0.X0e83> +
B(I, J-1)<0.X0483> +
B(I, J+1)<0.X0683>)/4

ENDDO
call update(2)
ENDDO

call update(1)

DO J = 2, N-1
DO I = 2, M-1
B(I, J)<0.X0403> = A(I, J)<0.X0403>

ENDDO
call update(2)
ENDDO
call update(1)
END

```

Figure 7: Reuse levels for the sample program

and its reuse is greater than w , then evicting the first reference in the next replacement will not degrade the overall LRU hit rate.

Proof. See [23].□

We can design an algorithm which sets the EM tag when accessing a reference if it knows its following reuse is sufficient far away. By Theorem 4, the replacement to the tagged block will not degrade the LRU hit rate, whereas the new replacement can possibly lead to one more hit by keeping in cache the block which would be evicted by the LRU policy. Accurately counting the number of distinct references mapped to a specific cache set is impossible at compile time when the starting memory addresses and size of arrays are unknown. Rather than count the number of references mapped into one specific set, we can estimate the total number of references in a loop nest 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 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 the cache size if it contains more than one level of loop nesting.

Now, we estimate the total data volume between a reference and its reuse that crosses loop nests. If it is greater than the cache size, then we predict that the number of distinct references mapped into a set between the reference and its reuse will be greater the degree of associativity. This intuition implies that Theorem 4 holds and leads to the algorithm in Figure 8.

The condition for setting the EM tags requires us to single out those references without temporal and spatial reuse in a nest. Unfortunately, there are few references in real applications that satisfy this condition. For instance, in nest 1 of Figure 3(a), all references have static spatial or temporal reuse, or both. We use the implementation described in Section 5.2 to resolve spatial reuse. At compile time, the EM tag of a reference is set when it has no temporal reuse in its nest. We encode its reference step into the corresponding instruction. At runtime, the EM tag for a reference with spatial locality is set by the runtime environment only after its spatial reuse is complete. In the program in Figure 3(a), we set the EM 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,

```

setEvictMeTag()
{
for each loop nest {
estimate the total data volume accessed in the nest
}

for each loop nest {
for each array reference  $a$  in the nest {
if  $a$  has no temporal and spatial reuse in the nest
&& ( it is not referenced in the following nests
|| the reuse is across nests whose estimated volume
is greater than cache size)
set the EM tag of reference  $a$ ;
}
}
}

```

Figure 8: Algorithm for setting EM tag

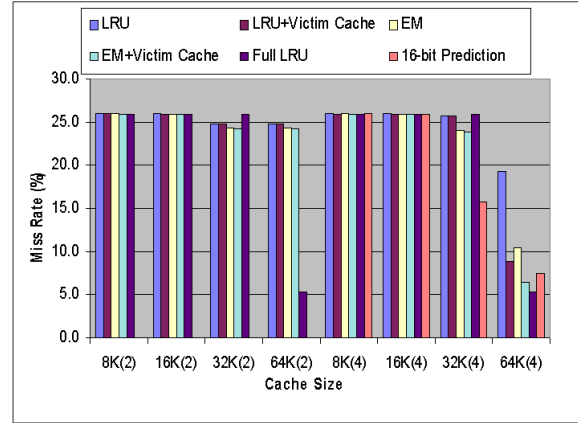


Figure 9: Vpenta

because these four references have no temporal reuses and the total data volume of each nest is estimated to be greater than the cache size.

6. Experimental Results

For our experiments, we use the Scale compiler infrastructure developed by ALI group at the University of Massachusetts. Scale accepts C and Fortran source code as input. The system translates the source code into an intermediate high and low-level mixed representation called Scribble. Scribble keeps the high-level program structure such as loop and array references and, at the same time, the low-level operations. The analyses proposed in Section 4 and Section 5 are applied to Scribble. The reuse levels or EM tags are annotated in Scribble through the Scale annotation tool. The back-end translates Scribble to C with annotations of reuse levels or EM tags. The annotations are implemented as special inline assembly instructions. We feed the C code with assembly inline to SimpleScalar 2.0, an architecture simulator [8]. We updated the SimpleScalar simulator to accept the special instructions. We also implement an 8-entry fully associative victim cache [13]. The victim cache is positioned between the first and second level cache. On a level one cache miss, the architecture checks the victim cache. When there is a hit to the victim cache, the hit entry is exchanged with the replaced victim in the level one cache, otherwise the replaced victim is put into the victim cache. In our simulation results, we do not show cycles because the cycle count in this version of SimpleScalar is accurate only when the memory system is lightly utilized [4]. The simulator reports accurate miss counts.

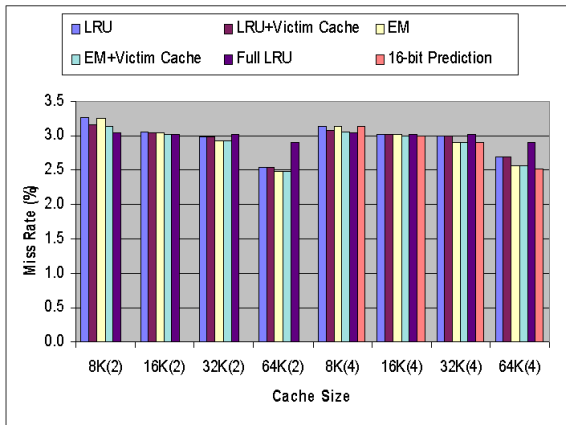


Figure 10: Appsp

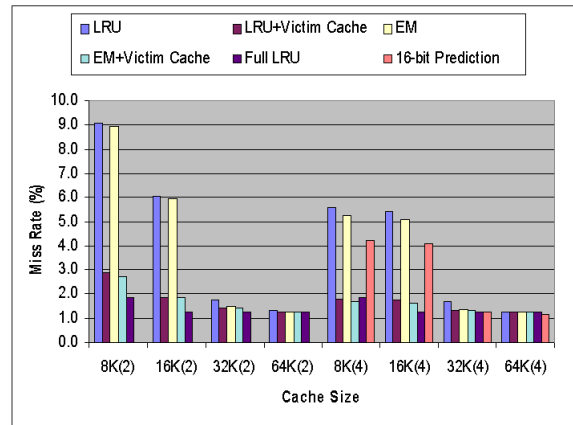


Figure 11: Tomcatv

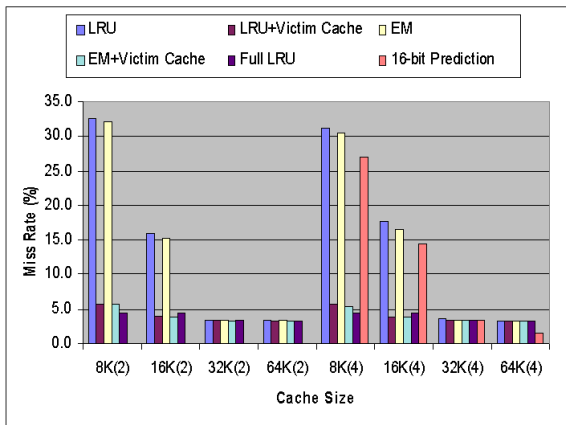


Figure 12: Swim

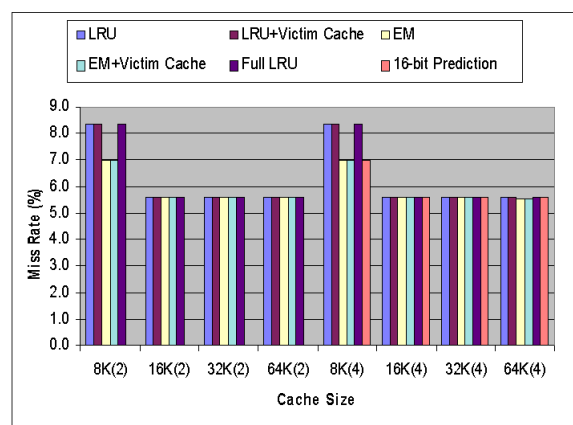


Figure 13: Jacobi

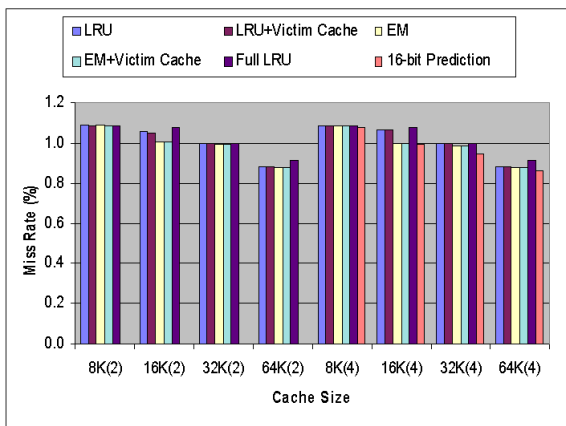


Figure 14: Erlebacher

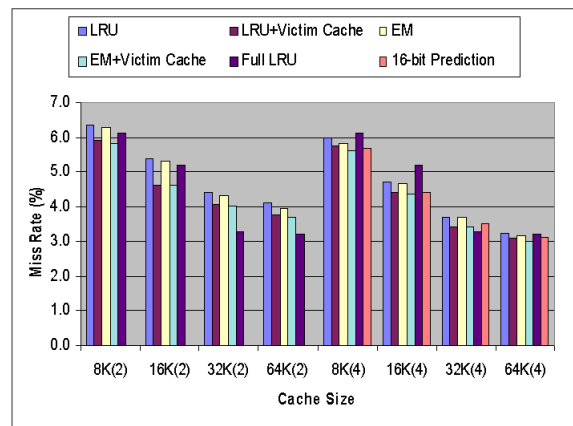


Figure 15: Arc2d

Program	2-way Miss Rate(%)		Percentage Improved	4-way Miss Rate(%)		Percentage Improved	EM(2)/ LRU(4)	Percentage of Static EM	Percentage of Dynamic EM
	LRU	EM		LRU	EM				
VPENTA	24.87	24.31	2.22	25.75	23.98	6.86	0.94	44.60	11.27
TOMCATV	1.77	1.49	15.74	1.70	1.37	19.62	0.88	10.79	2.01
SWIM	3.48	3.48	0.07	3.58	3.47	3.10	0.97	11.79	3.13
JACOBI	5.57	5.57	0.00	5.57	5.56	0.99	1.00	25.0	4.17
ERLEBACHER	1.00	0.99	0.85	1.00	0.99	0.85	0.99	8.91	1.25
ARC2D	4.41	4.34	1.48	3.70	3.68	0.44	1.17	6.00	2.23
APPSP	2.99	2.93	1.84	3.14	3.13	0.34	0.98	5.29	4.72

Table 3: Comparison of LRU and EM on Seven Programs for 32k Caches

The last two columns of Table 3 list the percentage of static tags set by compiler and the percentage of the tags used at run time for replacement decisions. Although, at runtime, fewer than 12% of EM tags are used for cache replacement decision, we observe more than 12% improvement in our simulations. In Figure 9 thru 15, we show the miss rates of LRU, LRU with victim cache, EM, EM with victim cache on 2-way or 4-way caches. We also measure the miss rates of LRU on fully-associative caches, and the 16-bit Prediction algorithm on 4-way caches. The cache sizes ranges from 8K to 64K and the cache line size is 32 bytes. We observe that EM provides up to 45% improvement in the number of misses in Vpenta for a 4-way 64K cache. This result is significant considering the minor architectural support we need. We improve the miss rate of Swim, Tomcatv, and Jacobi by 10-20% in the best cases. Appsp, Arc2d, and Erlebacher improve by 5-6% in best cases. EM never degrades the miss rate, although the miss-predictions we mentioned in Section 5 might cause a degradation.

Both EM and the 16-bit Prediction algorithm present significant improvements in certain cache configurations but very minor ones in others. Generally, EM and the Prediction algorithm reduce conflict misses. When the cache size is very big, there are few conflicts available for them to resolve. When the cache size is very small, the conflicts become so intense that no replacement algorithm can do well. The improvement is sensitive to the degree of associativity and cache line size, because those factors affect the distribution of conflict misses. Increasing the degree of associativity can reduce conflict misses and also give the EM algorithm and the Prediction algorithm more flexibility. We expect the EM and Prediction algorithms to increase their relative performance, as compared to LRU, in proportion to the increase in the degree of associativity when associativity is small.

An interesting comparison between the EM algorithm for a 2-way cache and the LRU algorithm for a 4-way cache is illustrated in column 8 in Table 3. In six out of seven programs, a 2-way EM is either much better than or very close to a 4-way LRU. The only exception is Arc2d, where the increase of associativity significantly reduces the miss rate. This result suggests that 2-way EM is probably a good idea considering the complexity and the hit cycle latency of a 4-way cache. The results of the 16-bit Prediction algorithm in 4-way caches are also shown in the figures. We notice that the 16-bit Prediction algorithm can further improve miss rate in some cases. In particular, for Vpenta and Swim at 64K, it accomplishes more than 20% improvement even when compared to a 4-way EM. This result means there is still some room for the compiler to improve, although we observe that the one-bit EM is sufficient in most cases.

Compared with fully associative cache, we notice that both victim cache and EM cache can attack certain capacity misses. There are a

number cases where the two strategies beat fully-associative LRU. However, we would rather attribute the results to the loose definition of capacity misses. Victim caches eliminate many misses in Tomcatv and Swim when cache size is small. However, in Jacobi, the victim cache has no effect at all, but EM shows 16% improvement at 8K. A further observation is that the two strategies can work together. For example, in Vepnta, the miss rate is reduced from 8.9%, when applying victim cache only, to 6.4%, when applying both. The victim cache by itself can outperform EM but putting the two strategies together always dominates using only one of them.

7. Conclusions

In this paper, we have developed a theoretical model for static compilation analysis to direct cache replacement and prove that it is at least as good as using the LRU algorithm. We present and implement 16 and 1-bit versions of our algorithm. We demonstrate that the 1-bit version is practical enough to implement in current set-associative level 1 caches. Furthermore, our simulation results show that our technique consistently improves miss rates when compared to the LRU replacement algorithm. In the future, we want to investigate applying this technique to non-numerical programs. We also want to use a cycle level simulator to more accurately characterize the performance benefits of approach.

8. Acknowledgments

Thanks to Chip Weems and David Culler for their insights on architectural trends. This work is supported by the National Science Foundation (NSF grants EIA-9726401, CCR-00-73401, CDA-9502639, and NSF CAREER Award CCR-9624209), Darpa (grant 5-21425), and Compaq. Any opinions, findings and conclusions or recommendations expressed in this material are the authors and do not necessarily reflect those of the sponsors.

9. 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] T. Austin and D. Bouger. Micro-30 simple scalar tutorial. Technical report, <http://www.cs.wisc.edu/mscalar/ss/tutorial.html>, 1997.
- [5] 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.
- [6] L. A. Belady. A study of replacement algorithms for a virtual-storage computer. *IBM Systems Journal*, 5(2):79–101, 1966.

- [7] 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.
- [8] D. C. Burger and T. M. Austin. The SimpleScalar tool set, version 2.0. *Computer Architecture News*, 25(3):13–25, June 1997.
- [9] 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.
- [10] J. Hennessy and D. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufmann Publishers, San Mateo, CA, 1995.
- [11] M. D. Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, Dec. 1988.
- [12] T. L. Johnson, M. C. Merten, and W. W. Hwu. Run-time spatial locality detection and optimization. In *Proceedings of the 30th International Symposium on Microarchitecture*, Research Triangle Park, NC, Dec. 1997.
- [13] 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.
- [14] 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.
- [15] 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.
- [16] 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.
- [17] 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.
- [18] 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.
- [19] 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.
- [20] M. Prvulovi, D. Marinov, Z. Dimitrijevic, and V. Milutinovic. A survey and reevaluation of performance. *IEEE TCCA Newsletters*, pages 8–17, 1999.
- [21] 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.
- [22] 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.
- [23] Z. Wang, K. S. McKinley, and A. L. Rosenberg. Improving replacement decisions in set-associative caches. Technical Report TR-01-02, University of Massachusetts, Mar. 2001. <http://ali-www.cs.umass.edu/>.