Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches

Dimitrios Stiliadis
Anujan Varma
Computer Engineering Department
University of California
Santa Cruz, CA 95064

Abstract

Victim caching was proposed by Jouppi [4] as an approach to improve the miss rate of direct-mapped caches. This approach augments the direct-mapped main cache with a small fully-associate cache, called victim cache, that stores cache blocks evicted from the main cache as a result of replacements. We propose and evaluate an improvement of this scheme, called selective victim caching. In this scheme, incoming blocks into the first-level cache are placed selectively in the main cache or the victim cache by the use of a prediction scheme based on their past history of use. In addition, interchanges of blocks between the main cache and the victim cache are also performed selectively. We show that the scheme results in significant improvements in miss rate as well as the number of interchanges between the two caches, for both small and large caches (4 Kbytes – 128 Kbytes). For example, simulations with four instruction traces from the SPEC Release 1 programs showed an average improvement of approximately 20 percent in miss rate over simple victim caching for a 16K cache with a block size of 32 bytes; the number of blocks interchanged between the main and victim caches reduced by approximately 74 percent. Implementation of the scheme in an on-chip processor cache is described.

1 Introduction

The last decade has seen an increasing gap between processor speed and the speed of the underlying memory hierarchy. This is the result of two contributing factors: (i) processor cycle time has been decreasing at a faster rate than memory access time, and (ii) design techniques such as pipelining and superscalar processing, common among modern RISC processors, have caused a dramatic reduction in the average number of cycles per instruction (CPI). These two factors together have resulted in a rapid increase in the cache-miss penalty in terms of the number of wasted instruction cycles. Influence of the memory hierarchy performance, of cache memory in particular, on program execution is therefore considerably stronger now than it was some time ago [7]. This trend is likely to continue with the new generation of processors such as the DEC Alpha, Intel Pentium, etc. Realizing the performance potential of these processors requires innovations in the cache memory subsystem.

Most of the current-generation single-chip microprocessors employ on-chip L1 (first-level) caches to provide fast access to instructions and data. The design of these on-chips caches involves a fundamental tradeoff between miss-rate and access time [3, 6]. A direct-mapped cache results in the lowest access time, but often suffers from high miss rates due to conflicts among memory references. Set-associative caches improve the miss rate at the expense of increasing the access time. Hill argued that direct-mapped caches often afford better performance in terms of effective memory-access time over set-associative caches [3]. In addition, as the processor cycle-time shrinks, single-cycle access to the cache can often be provided only by a direct-mapped configuration. Indeed, both the instruction and data caches in the DEC Alpha 21064 chip are implemented as direct-mapped to satisfy the 5 ns processor-cycle time [1].

The conflicts among memory references in a direct-mapped cache often result in significant increases in miss rate over set-associative caches, particularly when the cache size is small. This motivated researchers to investigate techniques to reduce conflict-misses in a direct-mapped cache without affecting its hit time (cache access time) [4, 5]. One such method, known as *victim caching*, was proposed by Jouppi [4]. In this approach, the main direct-mapped cache is augmented with a small fully-associative cache that is used to store the "victims" of replacements from the main cache. That is, upon every replacement, the block being discarded from the main cache is added to the victim cache, where it replaces the least recently used block. During each reference to the main cache, the victim cache is searched in parallel. In the event of a miss in the main cache that hits in the victim cache, the access is satisfied from the victim cache; a replacement is then effected by interchanging the corresponding blocks in the two caches.

Victim caching improves the effective memory access time by reducing the miss rate as seen by the second level of the memory hierarchy. This reduction in miss rate can be considerable for small caches [4].

This research is supported by NSF Young Investigator Award MIP-9257103 and NSF Grant No. MIP-9111241.

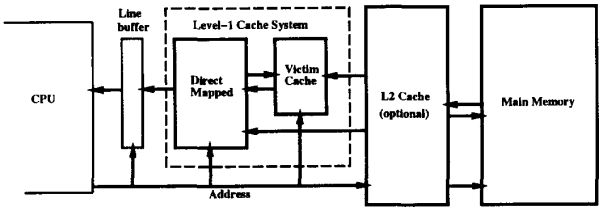

Figure 1: Memory hierarchy for the selective victim caching scheme.

As the size of the main cache is increased, however, the improvement decreases as a result of the small size of the victim cache. In addition, the time needed to exchange blocks between the main cache and the victim cache can offset some of the performance gain from the lower miss ratio.

In this paper, we propose and evaluate an improvement of victim caching, which we call *selective victim caching*. In this method, incoming blocks to the L1 cache are placed selectively either in the main cache or the victim cache using a prediction scheme based on their past history of use. Blocks that are less likely to be accessed in the future are placed in the victim cache instead of the main cache. Similarly, when a miss is serviced by the victim cache, the prediction algorithm is again applied to determine if an interchange of the conflicting memory items is required. Results from simulations show significant improvements in miss ratios for several traces; in addition, the method reduces the number of interchanges of blocks between the main cache and victim cache considerably, thus reducing the overhead introduced by them.

In our scheme, the decision to place an incoming block in the main cache or the victim cache is made with the aid of state information associated with cache blocks. These state bits provide information on the history of the block the last time it was in the main cache. This idea was first proposed by McFarling, who used the history information to exclude certain blocks from a direct-mapped cache, reducing cyclic replacements involving the same block [5]. This scheme, called *dynamic exclusion*, reduces conflict misses in many cases. Its main drawback, however, is that a wrong prediction results in an access to the next level of the memory hierarchy, offsetting some of the performance gain. In addition, the scheme is less effective with the large block sizes typical of current microprocessors. Selective victim caching, on the other hand, not only sustains but in some cases improves the miss ratios with increasing block size. Furthermore, the penalty of a wrong prediction in most cases is limited to an access to the victim cache, which takes at most one extra cycle. Finally, the effectiveness of the dynamic exclusion scheme is primarily limited to cases involving conflicts between two memory items, while selective victim caching can reduce the penalty for conflicts between more items, provided that the total space required does not exceed the available size of the victim cache.

The rest of this paper is organized as follows: Section 2 introduces and describes the selective victim caching idea. Section 3 presents a performance evaluation of the scheme by means of trace-driven simulations of four programs from the SPEC Release 1 benchmark suite. Section 4 discusses implementation alternatives for the scheme and describes an implementation for an on-chip cache where all state information is maintained within the CPU chip. Finally, some concluding remarks are given in Section 5.

2 Selective Victim Caching

In this section, we present the basic architecture of the memory hierarchy for application of selective victim caching and describe the algorithms involved. For simplicity, we describe the scheme as applied to an instruction cache. Extension of the scheme to data caches is straightforward.

The memory hierarchy is illustrated in Figure 1. The first-level (L1) cache consists of a direct-mapped main cache and a small fully-associative victim cache. A line buffer is included so that sequential accesses to words in the same cache block (line) do not result in more than one access to the cache, thus preventing repeated updates of state bits in cache. Upon the first access to a cache line, the entire line is brought into the line buffer in parallel with the execution in the CPU; subsequent accesses to words in the same line are satisfied from the line buffer and cause no update of state bits in cache. This allows the prediction algorithm to account for sequential references to words within a cache line as a single access to the cache line. Lines are replaced in the victim cache according to the LRU (least recently used) algorithm. The next lower level of the memory hierarchy can be an L2 cache or the main memory.

On every memory access, the line buffer, main cache, and the victim cache are searched in parallel. If the line is found in the line buffer, the instruction is fetched from there and no other action is necessary. Otherwise, three different cases must be considered:

1. *Hit in main cache:* If the word is found in the direct-mapped cache, it is delivered to the CPU and the entire line containing the accessed word is brought into the line buffer. This is no different than in the case of a simple direct-mapped cache. The only additional operation is a possible update of the state bits in cache used by the prediction scheme. We shall explain the function of the state bits later when we describe the prediction algorithm. The update can be performed in parallel with the fetch operation and introduces no additional delay.

2. *Miss in main cache, hit in victim cache*: In this case, the word is fetched from the victim cache into the line buffer and forwarded to the CPU. At the same time, the prediction algorithm is invoked to determine if the accessed block in victim cache is more likely to be accessed in the future than the block in main cache it is conflicting with. If the prediction algorithm decides that the

block in the victim cache is more likely to be referenced again than the conflicting block in the main cache, an interchange is performed between the two blocks; no such interchange is performed otherwise. In both cases the block in the victim cache is marked as the most recently used. In addition, the state bits used by the prediction algorithm are updated to reflect the history of accesses, as will be described later.

3. *Miss in both main and victim caches:* If the word is not found both in the main cache and the victim cache, it must be fetched from the next level of the hierarchy. This means that either the corresponding line in the direct-mapped cache is empty, or the new line conflicts with another line already stored there. In the first case, the new line is brought into the main cache and the victim cache is not affected. In the second case, however, the prediction algorithm must be applied to determine which of the two conflicting lines is more likely to be referenced in the future. If the incoming line is found to have a higher probability than the conflicting line in the main cache, the latter is moved to the victim cache and the new line takes its place in the main cache; otherwise, the incoming line is directed to the victim cache. Again, the state bits used by the prediction algorithm are updated, as will be described later.

The differences of our scheme from simple victim caching are in cases 2 and 3 above. In case 2, the conflicting blocks in the main cache and the victim cache are always exchanged in simple victim caching, whereas our scheme performs the interchanges selectively. Similarly, in case 3, the simple victim caching scheme always places incoming blocks in the main cache, whereas our scheme places them selectively in the main cache or the victim cache.

Having described the cache configuration, we can now describe the prediction algorithm employed. The prediction algorithm is invoked whenever the current access conflicts with a line stored in the main cache; the goal of the algorithm is to determine which of the two conflicting lines is more likely to be accessed in the future. The line with a higher probability of access in the future is placed in the main cache, and the other is placed in the victim cache. Thus, if the line in the victim cache is replaced in the future because of the small capacity of the victim cache, the impact would be less severe than the opposite choice. This is the basic idea behind our scheme.

The prediction algorithm is based on the dynamic exclusion algorithm proposed by McFarling [5]. The algorithm uses two state bits associated with every cache line, called the *hit bit* and the *sticky bit*. The hit bit is logically associated with a L1-cache block as it resides in L2-cache or main memory. A hit bit of 1 indicates that at least one hit to the corresponding block occurred while the block was in L1-cache *the last time*. A zero hit-bit indicates that the corresponding block was never accessed while it was in L1-cache the last time (and therefore less likely to be accessed more

```
Case 1: Access to line β, hit in main cache;
        (update hit and sticky bits)
        hit[β] ← 1; sticky[β] ← 1.

Case 2: Access to line β, hit in victim cache;
        Let α be the conflicting line in main cache;
        if sticky[α] = 0
            interchange α and β;
            sticky[β] ← 1; hit[β] ← 1;
        else
            if hit[β] = 0 then sticky[α] ← 0
            else
                interchange α and β;
                sticky[β] ← 1; hit[β] ← 0;
            end if
        end if

Case 3: Access to line β,
        miss in both main and victim;
        Let α be the conflicting line in main cache;
        if sticky[α] = 0
            move α to victim cache;
            transfer β to main cache;
            sticky[β] ← 1; hit[β] ← 1;
        else
            if hit[β] = 0 then
                transfer β to victim cache;
                sticky[α] ← 0
            else
                move α to victim cache;
                transfer β to main cache;
                sticky[β] ← 1; hit[β] ← 0;
            end if
        end if
```

Figure 2: The selective victim caching algorithm.

than once when the line is transferred to L1 cache the next time). In an ideal implementation, the hit bits are maintained in L2-cache or main memory and brought into L1-cache along with the corresponding block. When a block is removed from the L1-cache, the state of the corresponding hit-bit must then be updated in the L2-cache or main memory. We will later show an implementation in Section 4 that avoids such updates by maintaining the hit bits within the CPU chip; in this description of the prediction algorithm, however, we assume that the hit bits are associated with the L2-cache or main memory.

A sticky bit is associated with each block in the main cache of the L1 cache system. Its purpose is to

prevent thrashing by providing some inertia to a cache block. When a block, say α, is brought into the main direct-mapped cache, its sticky bit is set to 1. Every subsequent hit to the block α also refreshes the sticky bit to 1. On a reference to a conflicting block, say β, if the prediction algorithm decides not to replace α in the main cache, the sticky bit is now reset to zero. If a subsequent access conflicts with the block α while its sticky bit is zero, the block is replaced from the main cache.

The complete selective victim-caching algorithm is outlined in Figure 2. Three separate cases are considered: In the first case, a hit in the main cache simply sets the hit and sticky bits, if they are not already set. In the second case, the accessed block, say β, is in the victim cache. This implies a conflict between β and a block in the main cache, say α. In this case, the prediction algorithm is applied to determine if an interchange is to be performed. If the sticky bit of α is zero, implying that the line was not accessed since the previous conflict involving α, the new line β is given priority over α, causing an interchange. Likewise, if the hit bit of the line β is set, it is given higher priority over α, and the lines are interchanged. If the sticky bit of α is 1 and the hit bit of β is zero, the access is satisfied from the victim cache and no interchange is performed. The sticky bit of α is reset to zero in this case so that a subsequent conflict involving α without an intervening reference to it would result in α being moved out of the main cache.

Finally, case 3 of the algorithm presents the sequence of actions when an access misses in both the main and the victim caches. The sequence is similar to that of case 2, except that the destination of the incoming line is chosen as either the main cache or the victim cache. In the case of the former, the conflicting line in the main cache is moved to the victim cache before it is replaced.

Note that the penalty for a misprediction in this scheme is limited to an access to the victim cache and a possible interchange, assuming that the victim cache is large enough to hold the conflicting line between accesses.

The operation of the selective victim-caching algorithm can be illustrated with an example instruction reference sequence $(\alpha^m\beta\gamma)^n$ involving three conflicting lines α, β, and γ. The notation $(\alpha^m\beta\gamma)^n$ represents an execution sequence composed of two nested loops — the inner loop consisting of m references to the line α, followed by accesses to β and γ in the outer loop. The first access brings α into the main cache and both its hit and sticky bits are set by at most two references to it. When β is referenced, its hit bit is initially zero. Therefore it does not replace α in the direct-mapped cache and is stored in the victim cache instead. The conflict, however, causes the sticky bit of α to be reset to zero. When γ is referenced, its hit bit is zero, but the sticky bit of α is also zero. Hence, γ replaces α in the main cache. The line α is now transferred to the victim cache and its hit bit remains 1 because of the previous references. In the next cycle when α is referenced again, it is moved back to the main cache because of its hit bit remaining set. Thus, if the victim cache is large enough to fit both α and β or β and γ, only three references will be serviced by the second level cache. The total number of interchanges will be no more than $2n$. In the case of a simple prediction scheme without the victim cache the total number of misses would be $2n$, as the scheme can handle only conflicts between two lines. A simple victim cache with no prediction would be able to reduce the number of misses to the second level cache to 3, but would require a total of $3n$ interchanges during the execution of the outer loop. This brings out the advantage of selective victim caching over other schemes in dealing with conflicts involving more than two cache lines.

Trace program	Source language	Instruction references	Data references
gcc	C	100 M	21,428,034
li	C	100 M	23,664,933
spice2g6	FORTRAN	100 M	25,509,619
doduc	FORTRAN	100 M	32,403,178

Table 1: Benchmark programs used in the simulations.

3 Performance Evaluation

To evaluate the effectiveness of the selective victim caching idea, we performed extensive simulations using memory-access traces of several programs from the SPEC Release 1 benchmark suite. Because of constraints on the length of the paper, results from only four of the benchmarks are presented here. Results from other benchmarks can be found in [8].

Simulations were performed separately for instruction and data caches. In all cases, the size of the victim cache was fixed at 8 lines. The size of the direct-mapped main cache was varied from 2 Kbytes to 128 Kbytes. The line-size of the L1 cache system was chosen as 32 bytes for most simulations. However, we also studied the effect of varying the line size by performing a set of simulation runs for line sizes from 4 bytes to 64 bytes, keeping the size of the main cache fixed at 16 Kbytes.

A direct-mapped L2 cache with 8 times the capacity of the L1 main cache was employed in the simulations. The line size of the L2 cache was set equal to that of the L1 cache. The hit bits used by the prediction algorithm were stored in the L2 cache, but not in main memory. When a new line was brought into the L2 cache from main memory, its hit bit was initialized to zero. The hit bit was copied along with a line from L2 cache to L1, and the updated bit copied back to L2 upon replacement of the line in L1. To maintain the inclusion property in the case of data caches, lines in the L1 cache were invalidated when they were replaced in the L2 cache.

3.1 Program traces

To generate the memory-access traces needed, four programs from the SPEC Release 1 benchmark suite were traced on a SUN-4 workstation using the *qpt* trace-profiler program [4]. Table 1 lists the programs

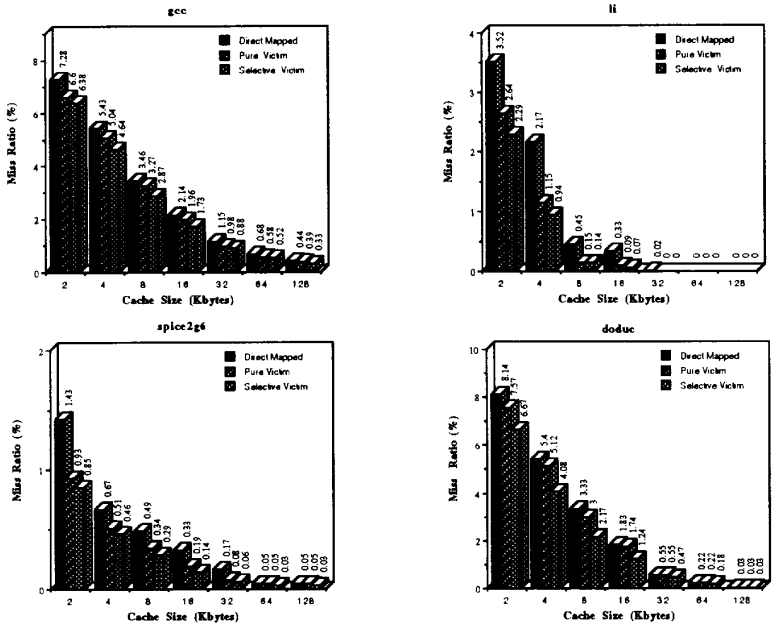

Figure 3: Miss ratio (%) for a direct-mapped instruction cache, a pure victim cache, and a selective victim cache configuration for different cache sizes with respect to four instruction traces (Line size = 32 bytes, size of victim cache = 8 lines).

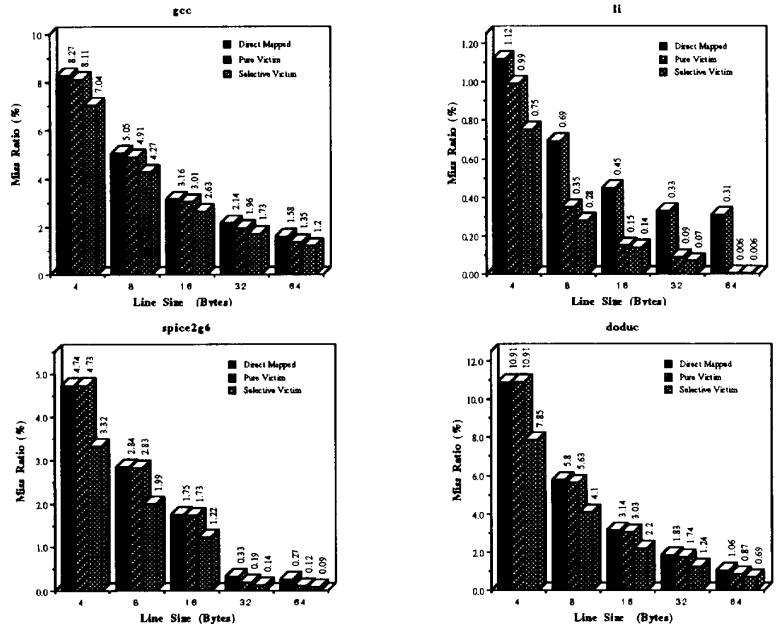

Figure 4: Miss ratio (%) for a direct-mapped instruction cache, a pure victim cache, and a selective victim cache configuration for different line sizes with respect to four instruction traces (Cache size = 16 Kbytes, size of victim cache = 8 lines).

used, along with their trace lengths. Two of the programs are written in C and two in FORTRAN. Since long traces are required to accurately simulate the behavior of large caches [9], we generated traces with approximately 100 million instruction references from each program. The number of data references in each trace is in the range of 20–30 million. We modified the *Dinero III* simulation package [2] to simulate our cache configurations.

3.2 Instruction cache simulations

In this section we present results from simulating the selective victim caching scheme in the instruction cache using the four program traces in Table 1. The results are compared with those for a direct-mapped cache and a simple victim cache configuration.

The cache miss-ratios for a direct-mapped cache a simple victim cache configuration and the selective victim cache are shown in Figure 3 as a function of the cache size. Note that the program trace of *li* was small enough to fit completely in a 64 Kbyte cache and generated no conflict misses beyond a cache size of 32 Kbytes. Only *gcc* has a miss rate of more than 1 percent in a direct-mapped cache of size 32 Kbytes.

The improvement in performance obtained by both pure victim caching and selective victim caching over a direct-mapped cache varies among the traces, depending on the size of their working sets and the frequency of the access conflicts the prediction scheme could eliminate. For benchmarks with small working sets and low miss ratios like *li*, the improvement for both schemes is high for cache sizes up to 32 Kbytes. For *li*, the reduction in miss ratio obtained by both schemes is as high as 95 percent for a 32K cache. In this case, the working set of the trace was found to be 794 lines (25,408 bytes). The instruction references produced a total of 16,243 conflict misses in the 32K direct-mapped cache; these were completely eliminated by the victim cache. As the cache size was increased beyond 32 Kbytes, however, the miss rate was dominated by the compulsory misses, and the improvement due to victim caching fell rapidly.

For benchmarks with relatively large working sets and high miss ratios like *gcc*, *spice2g6* and *doduc*, the reduction in miss ratio for the selective victim cache compared to the pure victim cache is as high as 35 percent for *spice2g6*, 30 percent for *doduc*, and 14 percent for *gcc*. For *spice2g6* with a 64K cache, the selective victim cache improves the miss ratio by as much as 40 percent, whereas pure victim caching gives almost no improvement. Even for small programs, selective victim caching provides a significant improvement compared to the pure victim cache when the cache is not large enough to fit the entire program.

Figure 4 shows the miss ratios as a function of the line size for all the different cache configurations. The size of the main cache is 16 Kbytes and the size of the victim cache is 8 lines. For most traces the reduction in miss ratio increases with the line size. This is contrary to the behavior of the dynamic exclusion scheme [5], where the miss rates increased with increasing line size. Thus, selective victim caching does not suffer from one of the main problems of prediction schemes. This outcome is the result of maintaining the same size for the victim cache in terms of the number of lines as the line size is increased, thus causing an increase in its effective capacity. This increase in capacity more than compensates for any degradation in the success rate of the prediction scheme with increasing line size. This does not, however, increase the complexity of the implementation of the victim cache, since its associativity remains the same. In most of the traces the improvement of the selective victim cache over the pure victim cache is almost steady irrespective of the line size. In the case of the small programs like *li*, a different behavior is observed because the compulsory misses dominate the miss ratio.

An overhead introduced by the victim caching is the time needed to interchange cache lines between the main cache and the victim cache. In the case of pure victim caching, an interchange is performed on every hit to the victim cache. One of the goals of selective victim caching is to reduce the number of these interchanges, which have an influence on the effective memory access time. A simple expression for the effective memory access time with victim caching can be derived as follows: Let r be the total number of references, v the number of main-cache misses that are serviced by the victim cache, and m the number of misses from the entire L1-cache system. Furthermore, let p_v be the penalty for accessing the victim cache (in the event of a miss in the main cache), p_2 the penalty for accessing the next level of the hierarchy, and p_i the penalty for an interchange. If i is the number of interchanges performed, the average memory access time is given by

$$\frac{(r-m) + mp_2 + vp_v + ip_i}{r}.$$

The penalty for a hit in the victim cache with careful implementation may be one cycle. In the case of pure victim caching the number of interchanges i is identical to the number of hits to the victim cache v; for selective victim caching, however, i can be much less than v.

Figure 5 presents the number of interchanges for both pure victim caching and selective victim caching for the same four instruction traces presented in Figure 3. The improvement was more than 40 percent for most of the traces and cache sizes, and over 90 percent in several cases. When the line-size is large, depending on the implementation, the interchange operation may need several clock cycles. Therefore, the reduction in the number of interchanges can reduce the overhead of victim caching considerably. Thus, by improving both the miss ratio and the number of interchanges, selective victim caching can significantly improve the performance of the first-level cache system.

3.3 Data cache simulations

We also applied the selective victim-caching scheme to a first-level data cache. The same four traces in Table 1 were used to evaluate the scheme. The same prediction scheme and memory hierarchy were used in the simulations. The write policy implemented was write-back with write-allocate. To maintain the multilevel

inclusion property, lines in L1 cache were invalidated when they were replaced in the L2 cache.

Figure 6 shows the miss ratios for data caches for a direct-mapped cache, a pure victim cache and a selective victim cache configuration as a function of the cache size. Selective victim caching produced significant improvements in performance for two of the traces — *spice2g6*, and *doduc*. For *li*, the reduction in miss ratio is already high with pure victim caching (as high as 95 percent for large caches), and our scheme did not produce any additional improvement in performance. In fact, for *li* the relative performance improvement of the both the victim cache and the selective victim caching was found to increase with the cache size. The reason is the large number of conflict misses produced by the trace in a direct-mapped cache, which were reduced significantly by the victim cache. Both the pure victim caching and the selective victim caching schemes were successful in removing a large percentage of these misses.

As can be seen from Figure 6, the selective victim caching actually degraded the performance in several cases in comparison to the pure victim cache. The reason for this behavior is apparent from the nature of the programs used. Programs that employ static allocation of data and regular data structures, such as *spice2g6*, showed significant improvements as a result of the use of the prediction algorithm. Both *spice2g6* and *doduc* are FORTRAN programs that use regular data structures and static allocation. The main data structures in both programs are arrays. The prediction algorithm is able to resolve a large number of conflicts in these cases without an access to the second level. In the case of programs that use dynamic memory allocation and extensive use of pointers, however, the conflicts were much harder to resolve by the prediction algorithm. Examples are *gcc* and *li*. Both *gcc* and *li* are C programs with extensive use of lists and hash tables. The conflicts in these programs were much less regular and difficult to predict.

The improvement in the number of interchanges of cache lines between the main cache and the victim cache was also smaller for the data traces as compared to the instruction traces. For example, for an 8 K cache with a line size of 32 bytes, the reduction in interchanges varied from approximately 32 percent for *doduc* to almost no improvement for *spice2g6*. Detailed results can be found in [8].

4 Implementation

The selective victim caching scheme requires two state bits to keep track of the history of a cache line — the *sticky bit* and the *hit bit*. In this section, we discuss how this state information can be maintained within the memory hierarchy.

The sticky bit is logically associated with a line in the main cache. It is therefore natural to store it in the direct-mapped cache as part of each line. The hit bit, on the other hand, is logically associated with each line in the main memory. Thus, in a perfect implementation, the hit bits must be stored in the main memory. This approach is impractical in most cases. Therefore, we must resort to an approximate implementation.

If the memory hierarchy includes a second-level cache, it is quite feasible to store the hit bits alongside the lines in L2 cache, as suggested by McFarling [5]. When a line is brought into the L1 cache from L2, a local copy of the hit bit is stored along with the line in L1 cache. This eliminates the need to access the L2 cache every time the hit bit is updated by the prediction algorithm. When the line is replaced from L1 cache, the corresponding hit bit is copied into L2. This is the scheme used in our simulations in section 3.

A problem with this scheme is that multiple items in main memory are forced to share the same hit bit in L2 cache. Thus, when a line is replaced from L2 cache, all its state information is lost, reducing the effectiveness of the prediction scheme. Each time a line is brought into L2 cache from main memory, its hit bit must be set to an initial value. For a specific access sequence different initial values may produce different results. With large L2-caches, however, the impact of this problem may be small.

A second problem that arises from maintaining the hit bits in L2 cache is the overhead of copy-back whenever replacements occur in the L1 cache. This overhead may be small in a data cache, because copy-backs of dirty lines must be performed in any case. In the case of an instruction cache, however, the overhead is significant.

A third approach is to maintain the hit bits within the CPU itself, alongside the L1 cache. Since the size of this state storage must be kept small, some kind of hashing must be employed to share the hit bits among the cache lines. We now describe such an approach and evaluate its performance.

In our approach, an array of hit bits called the *hit array* is maintained as part of the L1 cache system. Each memory line is mapped to one of the bits of these array. The length of the hit array is inevitably smaller than the maximum number of lines that can be addressed. Hence, more than one line will be mapped to the same hit bit, causing some randomization to be introduced in the prediction process. Although this can potentially degrade the performance of selective victim caching, results from our simulation showed almost no such degradation even with a small hit array.

The actual implementation of the L1 cache system with hit array is shown in Figure 7. The line buffer is not shown. A sticky bit is maintained with each line in the main cache. No state bits are needed in the victim cache. The hit bits are kept in the hit array, addressed by a part of the memory address as shown in Figure 7. The size of the hit array is chosen as a multiple of the number of lines in the main cache. That is,

$$\text{Size of hit array} = \text{Number of lines in main cache} \times H,$$

where H determines the degree of sharing of the hit bits among the blocks in main memory. We assume that H is a power of two, $H = 2^h$. Then, the hit array can be addressed by the block address concatenated with the least significant h bits from the tag part of the address, as shown in Figure 7. A larger H implies less interference among conflicting lines for hit bits.

U-M-I
DUE TO LACK OF CONTRAST, GRAPHS DID NOT REPRODUCE WELL.
GRAPHS FOLLOW SAME SEQUENCE AS LEGEND

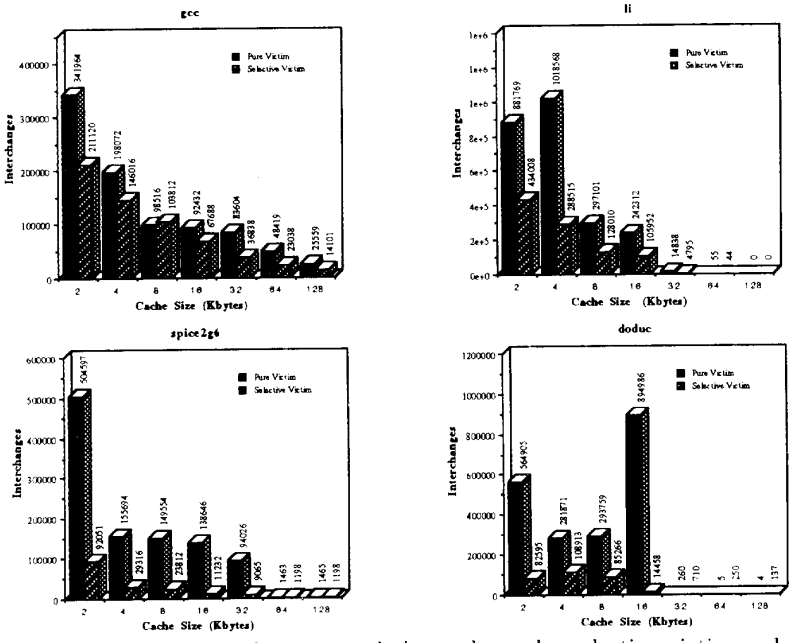

Figure 5: Number of word interchanges for a pure victim cache and a selective victim cache configuration as a function of the cache size with respect to four **instruction** traces (Line size = 32 bytes, size of victim cache = 8 lines).

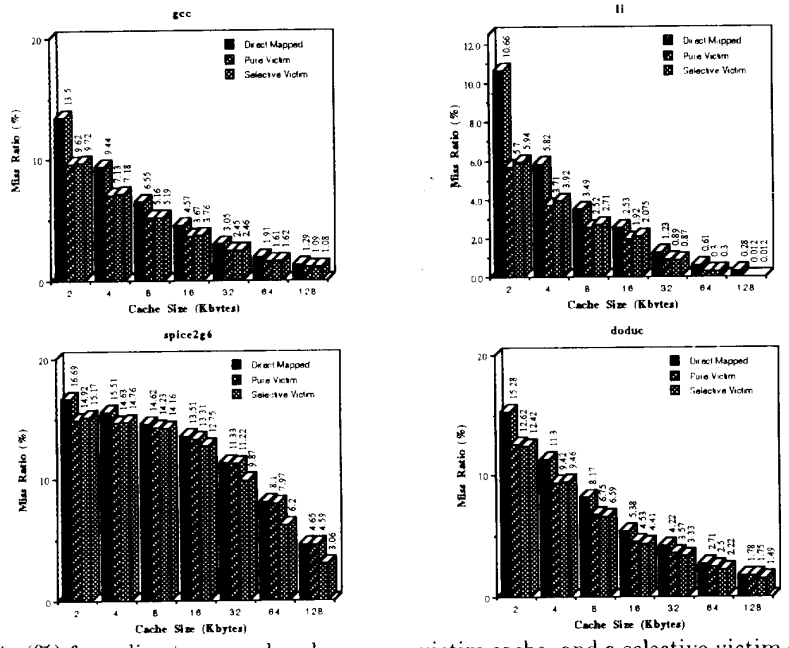

Figure 6: Miss ratio (%) for a direct-mapped cache, a pure victim cache, and a selective victim cache configuration for different cache sizes with respect to four **data** traces (Line size = 32 bytes, size of victim cache = 8 lines).

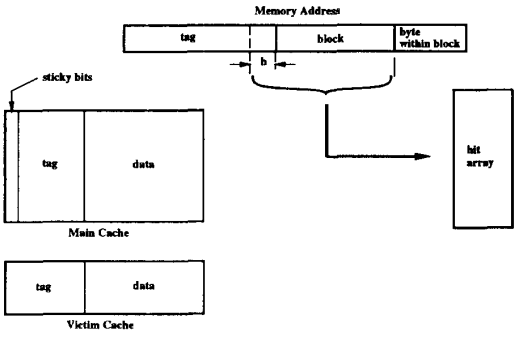

Figure 7: Implementation of the hit-array scheme.

If H is chosen identical to the ratio of the size of the L2 cache to the size of the L1 main cache, then the effect is similar to maintaining the hit bits in L2 cache, except that copying of bits between the two levels is avoided.

To evaluate the effect of sharing of hit bits resulting from a small hit array, we implemented the scheme in our cache simulator and ran simulations for the cases considered in section 3. We chose H as 4. With the ratio of L2 to L1 cache size of 8, this corresponds to one hit bit shared by two lines in L2 cache. When a new line is brought into the L1 cache, the current state of the hit bit selected by its address in the hit array is used as its hit bit, although this bit may represent the state of a different line in memory at that time. Simulation results with the hit-array implementation are shown in Figures 8 and 9. The plots in Figure 8 are for the instruction cache. Note that the effect of sharing of bits in the hit array is barely noticeable in the plots. The total storage required for the hit array is also modest — 256 bytes for a 16K cache with 32-byte lines.

Figure 9 shows the same comparison for a data cache. For the hit-array scheme, no L2 cache was simulated. This actually led to an improvement in performance as a result of the elimination of invalidations from the L2 cache to L1. Such invalidations are caused by the need to satisfy the inclusion property. The hit-array allows the selective victim caching scheme to be implemented without an L2-cache, thus improving the effectiveness of the prediction scheme slightly. Clearly, the effect of invalidations on the prediction scheme can also be reduced by designing the L2-cache as set-associative.

5 Conclusions

In this paper we presented a method for improving the performance of victim caching. The method selectively places data in the main cache or the victim cache based on a prediction algorithm that determines how likely they are to be accessed in the future. Results from simulations of four programs from the SPEC Release 1 benchmark suite showed significant improvements when the scheme was applied to a first-level instruction cache. For a cache size of 16 Kbytes, selective victim caching produced an average improvement in miss rate of approximately 20 percent over pure victim caching. The improvement was smaller when the idea was applied to a data cache, averaging only 4 percent over pure victim caching; for some of the traces, however, the improvement was much higher.

For both instruction and data caches, the number of interchanges of blocks between the main cache and the victim cache decreased considerably as a result of selective victim caching. The average reduction for the four traces was approximately 74 percent for a 16K instruction cache.

Because both the miss rate and the number of interchanges affect the average access time of the memory hierarchy, reducing either significantly can potentially lead to considerable improvements in the memory-access performance. Future work includes techniques to improve the effectiveness of the prediction algorithm for data accesses.

References

[1] *DECchip 21064-AA Microprocessor Hardware Reference Manual*, Digital Equipment Corporation, October 1992.

[2] M. D. Hill. *Aspects of cache memory and instruction buffer performance*, Ph.D. Dissertation, Computer Science Department, University of California, Berkeley, November 1987.

[3] M, D. Hill, "A case for direct-mapped caches," *IEEE Computer*, December 1988, pp. 25–40.

[4] N.P. Jouppi. "Improving direct–mapped cache performance by the addition of a small fully-associative cache and prefetch buffers," *Proc. 17th Annu. Int'l. Symp. Computer Architecture*, May 1990, pp. 364–373,

[4] J. R. Larus. "Efficient program tracing," *IEEE Computer*, May 1993, pp. 52–61.

[5] S. McFarling. "Cache replacement with dynamic exclusion," *Proc. 19th Annu. Int'l. Symp. Computer Architecture*, May 1992, pp. 192–200.

[6] S. Przybylski, M. Horowitz, and J. Hennessy, "Performance tradeoffs in cache design," *Proc. 15th Annu. Int'l. Symp. Computer Architecture*, June 1988, pp. 290–298.

[7] A. J. Smith. "Cache memories," *Computing Surveys*, September 1982, pp. 473–530.

[8] D. Stiliadis and A. Varma, "Selective victim caching: A method to improve the performance of direct-mapped caches," Technical Report UCSC–CRL–93–41, October 1993.

[9] D. W. Wall, A. Borg, and R .E. Kessler, "Generation and analysis of very long address traces," *Proc. 17th Annu. Int'l. Symp. Computer Architecture*, May 1990, pp. 270–279.

U-M-I
DUE TO LACK OF CONTRAST, GRAPHS DID NOT REPRODUCE WELL.
GRAPHS FOLLOW SAME SEQUENCE AS LEGEND

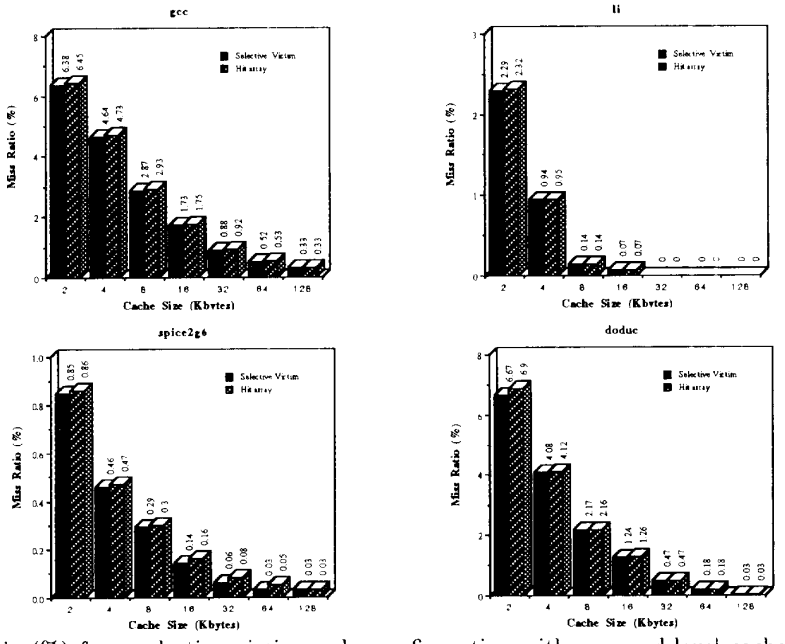

Figure 8: Miss ratio (%) for a selective victim cache configuration with a second-level cache and the hit-array implementation with respect to four **instruction** traces (Line size = 32 bytes, size of victim cache = 8 lines, $H = 4$).

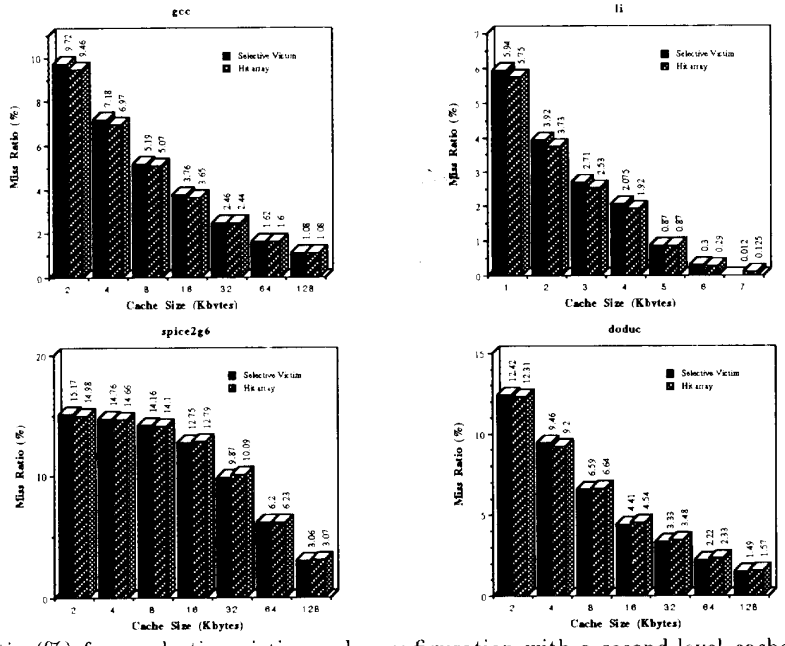

Figure 9: Miss ratio (%) for a selective victim cache configuration with a second-level cache and the hit-array implementation with respect to four **data** traces (Line size = 32 bytes, size of victim cache = 8 lines, $H = 4$).