Improving Performance of an L1 Cache With an Associated Buffer

Vijayalakshmi Srinivasan
Electrical Engineering and Computer Science,
University of Michigan,
1301 Beal Avenue,
Ann Arbor, MI 48109-2122,USA.
E-mail: sviji@eecs.umich.edu

February 12, 1998

Contents

1 Introduction **2**

2 Related Work **3**

3 NT-victim Cache **6**
3.1 Cache Organization 6
3.2 NT-victim Algorithm 7
3.3 Cache Replacement Policies 8
3.3.1 Pseudo-optimum Replacement Policy 8
3.3.2 NT-victim Cache Replacement Policies 11

4 Experimental Setup **13**
4.1 Benchmarks 13
4.1.1 Description 14
4.1.2 Miss Ratio for Direct-Mapped Cache 15
4.2 Performance Metrics 16
4.3 Simulator 17

5 Results **18**
5.1 Miss ratio 18
5.2 Swaps Between Cache A and Cache B 22

6 Conclusions and Future Work **24**

7 Acknowledgments **26**

List of Tables

1 *pseudo-opt* replacement - Miss Ratio 10
2 Benchmark Descriptions 15
3 Miss Ratio 20
4 Miss Ratio without Swaps 22
5 Swaps between Cache A and Cache B 23

List of Figures

1 NT-victim Cache Organization 6
2 Misses using the pseudo-opt policy 10
3 Miss Ratio *vs.* Cache Size 15
4 Miss Ratio 19

5 Percent NT replacements 21

Abstract

Memory access latencies are much larger than processor cycle times, and the trend has been for this gap to increase over time. Cache performance becomes critical in bridging this gap. However, since it is difficult to make a cache both large and fast, cache misses are expected to continue to have a significant performance impact. Victim caching, proposed by Jouppi [7], is an approach to decrease the miss ratio of direct-mapped caches without affecting their access time. NTS caching, proposed by Rivers [10] is a multilateral cache design scheme that improves performance of first-level(L1) caches based on the temporal locality of the reference pattern. We propose an improvement of these schemes, which we call NT-victim caching. Taking the lead from NTS design we have a bilateral L1 cache, having a main cache (cache A) and a small fully associative buffer (cache B). Cache B is similar to a victim buffer, and holds the victim block replaced by a miss. A cache block is temporal if after it is brought into cache, some word in that block is accessed more than once before the block is replaced. Unlike Victim caches a block that is hit in cache B is swapped with a block in cache A only if it is temporal, and in most of our replacement strategies temporal blocks are less likely to be selected for replacement than non-temporal blocks. Every cache block loaded into L1 cache is monitored for temporal behavior by a hardware detection unit. Our results show that this design reduces the number of swaps between cache A and cache B, relative to the Victim cache, yet gives a comparable miss ratio.

1 Introduction

Over the last decade, processor cycle time has been decreasing much faster than main memory access time. In addition, aggressive superscalar machines target 16 instructions per clock cycle [9]. With these effects, given that about a third of a program's instruction mix are memory references, data cache performance becomes even more critical. However, data caches are not always used efficiently. In this study we focus on data cache performance optimization.

The design of an on-chip first-level cache involves a fundamental tradeoff between miss ratio and access time. Tyson et al. [18] show that cache block placement and replacement strategy can be improved using characteristics of individual load instruction. In numeric programs there are several known compiler techniques for optimizing data cache performance. However, integer programs often have irregular access patterns that are more difficult for the compiler to optimize. Our work concentrates on performance improvement of L1 caches for integer benchmarks.

To have low access times, the main L1 cache is usually designed to have low associativity[5]. A direct-mapped cache results in the lowest access time, but often suffers from high miss rate due to conflicts among memory references. In order to increase data cache effectiveness we investigate methods of data cache management, where we control the movement and placement of data in the hierarchy based on the data usage characteristics.

Prior work shows that the performance of a direct mapped L1 cache can be improved by adding a small fully associative buffer in the same level, the primary examples being Victim cache, Assist cache, and NTS cache. Victim cache [7] reduces conflict misses of a direct mapped cache by adding a small fully associative victim buffer that stores blocks that are recently replaced in the main cache, i.e., recent "victims". Assist cache [8] reduces block interference and cache pollution by loading blocks from memory into a small fully associative buffer. Blocks are promoted to the main cache only if they exhibit temporal locality. NTS cache [10] uses a small fully associative buffer in parallel with the main cache for storing and managing blocks containing *non-temporal* data. This reduces block conflicts and cache pollution and separates the reference stream

into *temporal* and *non-temporal* block references.

In this paper we analyze the above schemes and present a new scheme, which we call NT-victim caching. In NT-victim cache, the main cache is associated with a small fully associative buffer. Hardware determines data placement based on dynamic reference behavior. Our analysis shows that the victim buffer is very effective in reducing conflict misses. However, Victim caches swap a pair of blocks between the main cache and the victim buffer on *every* hit in the buffer. To achieve miss rates comparable to the Victim cache we retain the basic characteristics of the victim buffer for our small fully associative buffer. However, to reduce the number of swaps between the main cache and the fully associative buffer we exploit temporal locality in a fashion similar to NTS scheme. We use a hardware detection unit to tag each cache block temporal or non-temporal. A cache block is *temporal* if after it is brought into cache, some word in that block is accessed more than once before the block is replaced. Whenever a *temporal* block is hit in the fully associative buffer it is swapped with a block in the main cache. However, *non-temporal* block hits in the buffer do not result in any movement between the caches. In addition we exploit temporality when making replacement decisions. Our results show that using temporal information reduces the number of swaps compared to the Victim cache, yet gives a comparable miss ratio.

The rest of the paper is organized as follows. We review related work in Section 2. Section 3 describes our technique for improving L1 cache performance. We discuss the performance metrics and present details of the simulation and the benchmarks used for this study in section 4. We present results in section 5, and conclusions and future work in section 6.

2 Related Work

Optimizing small caches for numerical workloads and reducing conflict misses have received extensive examination. One approach proposed by Jouppi [7] is *Victim Caching*, where a small fully-associative "victim buffer" is introduced between the L1 direct-mapped(DM) cache and main memory. Blocks replaced from the L1 cache are stored in the victim buffer. References that are served by the victim buffer have a very low

miss penalty. Since the victim buffer is fully-associative, blocks that reside in the victim buffer may be associated with any set of the DM cache. Consequently, the victim buffer is ideally suited to relieving conflict misses in hot sets. However, in this scheme whenever a reference hits in the victim buffer, the hit block is swapped with a block in the main cache. This causes ping-pong swaps if two conflicting blocks are referenced alternately. For example, consider an access pattern $(\alpha\beta)^n$, where α and β are two conflicting blocks. In a design with a direct mapped cache A, and a fully associative cache B (victim buffer), we see that α and β are swapped between caches A and B on every reference. Hence this scheme can potentially lead to an excessive number of swaps.

Agarwal and Pudar [1] present a *Column Associative Cache* to reduce conflict misses by dynamically applying a second hashing function during conflicts to place some cache lines in a different set. This scheme relies on the effectiveness (orthogonality) of the two hash functions. The latency of checking for a cache hit can increase depending on the number of links that must be followed to fetch the block. A small hash table lookup time is critical for good performance of this scheme.

Selective Victim caching proposed by Stiliadis and Varma [14] places incoming blocks selectively in the main cache (cache A) or a small victim buffer (cache B) using a prediction scheme. This scheme swaps a block from cache B to cache A based on its history of use. We take the lead from this scheme in doing *selective* swaps between cache A and cache B. However, we improve this scheme by taking into account the temporality of a block when making swap decisions. For instruction caches, Selective Victim caching shows dramatic performance improvement compared to Victim caching; however, it shows no performance improvement for data caches.

Bennett and Flynn [3] propose *Prediction Caches* that use a history of recent cache misses to predict future misses. Prediction caches combine the features of prefetching and Victim caching. This scheme, like Victim caches, helps alleviate hot spots in cache. However, none of the above four studies addresses the problem of cache pollution.

Unlike the above methods, the HP-7200 *Assist Cache* [8] places a conventional L1 cache in parallel with a small fully associative cache, guaranteeing a one-cycle lookup in both units. Blocks from memory are first loaded into the assist cache (cache B) and

are promoted into main cache (cache A) only if they exhibit temporal locality. We use this idea of temporality in our design when we swap blocks between cache B and cache A. In the Assist Cache, all the incoming blocks are placed in the small cache B and the LRU block from cache B is replaced to make room for the incoming block. This could potentially replace blocks before they have a chance to become temporal. We improve this scheme by placing all incoming blocks in cache A and giving victims of replacement a longer lifetime in cache B, so they have a greater chance to become temporal.

The *NTS cache* proposed by Rivers and Davidson [10] supplements the conventional direct-mapped cache (cache A) with a parallel fully associative cache (cache B). This scheme separates the reference stream into *temporal* and *non-temporal* block references. Blocks are treated as *non-temporal* until they become *temporal*. Cache blocks that are identified as *non-temporal* when they are replaced are allocated to cache B on subsequent requests. This decreases the conflicts between *temporal* and *non-temporal* blocks for a place in the same cache. However, NTS caching does not allow swaps between cache A and cache B. This becomes critical when we have conflicts among *temporal* blocks or conflicts among *non-temporal* blocks. This degrades the performance by not utilizing cache space that might be available in L1. Although we use a similar mechanism to tag blocks as *temporal*, we improve this scheme by allowing both *temporal* and *non-temporal* blocks to reside in cache A and in cache B.

Johnson and Hwu [6] have investigated a technique for dynamic analysis of program data access behavior, which is then used to guide the placement of data within the cache hierarchy. In this scheme, infrequently accessed data bypass the cache when they conflict with much more frequently accessed data. This reduces conflicts and alleviates cache pollution.

All the above methods describe a modification of the cache design to enhance the performance of an L1 cache. Tyson et al [18] present a detailed characterization of data cache behavior for individual load instructions and describe a scheme to selectively allocate cache lines according to the characteristics of the load instructions. Their results suggest that data reference behavior plays a critical role in cache block placement and replacement decisions. In our scheme we control the movement and placement of data

in the hierarchy based on the dynamic data usage characteristics.

3 NT-victim Cache

In this section we present the basic architecture of the memory hierarchy for NT-victim caching and describe the algorithms involved.

3.1 Cache Organization

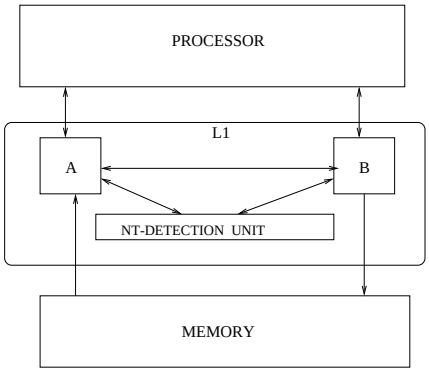

Figure 1: NT-victim Cache Organization

A block diagram of the NT-victim cache organization is shown in Figure 1. The main cache (cache A) is a direct-mapped cache. Cache B is a small fully-associative cache in the same level, used for storing some combination of *non-temporal* blocks and victim blocks. The next lower level of the memory hierarchy can be either a conventional L2 cache or main memory. Cache B is small compared to cache A because it is a fully associative structure; if it is large, the cost of the tag search may offset the gain in performance.

The NT detection unit is a hardware bit-map structure similar to the one described in [10]. It is attached to both cache A and cache B and is laid out as a matrix of bits with as many columns as the number of words in a cache block and as many rows as the number of sets in cache A plus the number of blocks in cache B, i.e., the total number of blocks in A and B. Each cell of this matrix keeps track of the usage count of each word of the block. On every memory access, the usage count of that word is incremented using the word offset of the block accessed.

For simplicity, we assume the same cache block size across cache A, cache B, and the next level of hierarchy. To avoid aliasing problems and cache flushing after a context switch, our design requires that both cache A and cache B be physically indexed.

On a reference, caches A and B are searched in parallel. A hit in cache A is no different from a hit in a conventional L1 cache. If the access results in a miss in cache A, but a hit in cache B, it is simply fetched from cache B, and counted as a hit for the purpose of performance evaluation.

We tag each block as either *temporal* or *non-temporal* following the ideas in [10]. The *lifetime* of a block refers to the time interval that the block spends in cache from one of its allocations (in A or B) until its next replacement from L1; a block may thus have several lifetimes.

- *Temporal Block*: A block is considered *temporal*, i.e., its T bit is set, if during its lifetime, at least one of its words is re-referenced. Such blocks exhibit the property of temporal reuse and are also referred to as T-blocks.
- *Non-Temporal Block*: A block leaving the cache is considered *non-temporal*, i.e., its T bit remains reset, if during its lifetime, none of its words is re-referenced. Such blocks are also referred to as NT-blocks.

In this study, unlike [10], we have assumed that NT/T information is not carried into the next level of the hierarchy. Every block begins each lifetime in the L1 cache as an NT-block; it becomes a T-block as soon as it exhibits temporality during this lifetime. We exploit this NT/T information when we make swap and replacement decisions.

3.2 NT-victim Algorithm

NT-victim cache differs from Victim cache in its replacement policy and its decision criteria for swapping blocks between cache A and cache B. (In the rest of the paper we refer to swap between cache A and cache B simply as *swap*). In Victim caching, when a block is hit in cache B, it is swapped with a block in the corresponding set of cache A; on a cache miss, the LRU block is chosen for replacement, irrespective of its access patterns.

We now describe the NT-victim caching algorithm in detail. Since the algorithm does not depend on the associativity of cache A, we present the algorithm for an M-way associative cache A. On every memory access, both cache directories are accessed in parallel. Assume that the access is to a block b and that it maps to set s, whose current LRU block is c. The possible cases are:

- *Hit in cache A*: If b is found in cache A, the actions are no different from any conventional L1 cache hit. b becomes the most recently used block of set s. If this reference causes a word reuse in b then the T bit of the block is set. This update does not introduce any additional delay.
- *Hit in cache B*: If b is found in cache B, we incur no additional delay. The block b may have its T bit set already. Otherwise if this reference causes a word reuse in b, its T bit is set. We have two cases to consider depending on whether b is a *temporal* block or not.
  - b is a *temporal* block: b is swapped with c and becomes the most recently used block of set s. Similarly c becomes the most recently used block in cache B. A block's T-bit remains with it while it resides in the L1 structure.
  - b is a *non-temporal* block: We simply update cache B to make b the most recently used block.
- *Miss in both caches*: b is brought into the main cache from the next level of memory. If s has fewer than M blocks, we simply make b the most recently used block in s. If not, it replaces c (the "victim" of this miss),which then moves to cache B. If B has an empty block to hold c, we simply make c the most recently used block in cache B; otherwise (cache B is full) we need to select a block in B for replacement. In the following subsection we describe several replacement policies that we investigate in this study.

3.3 Cache Replacement Policies

3.3.1 Pseudo-optimum Replacement Policy

Optimal replacement policies based on complete knowledge of future references provide a standard against which we can compare the performance of the implementable

policies. In order to understand the effect of replacement decisions on performance improvement, we analyze the miss ratio for our cache configuration using a *pseudo-opt* replacement policy. Victim cache, NTS cache and Assist cache, each use a simple LRU replacement strategy. In our simulations, the *pseudo-opt* replacement policy typically produced a 50% reduction in miss ratio, compared to these designs. Although *pseudo-opt* itself is not implementable, this result suggests that there is great potential for improving the performance of these cache designs by discovering and using better replacement policies. We use the *pseudo-opt* experiments to first understand the effectiveness of replacement decisions on cache performance and then use these results to help guide the search for good replacement algorithms for our NT-Victim cache design.

Our *pseudo-opt* replacement algorithm is an adaptation of Belady's [2] MIN algorithm, which is an optimum replacement policy for a fully associative cache. On a miss, the block to be replaced is the one whose next reference is farthest in the future. This choice clearly minimizes the number of misses in a cache set.

In our scheme, we have two caches A and B with different associativity. We allow free movement of blocks between them, but maintain disjoint contents in them at any particular time. We study the following extension to Belady's MIN algorithm. Consider an M-way associative cache A and a fully associative cache B of size K. On a miss, if there is an empty block in the corresponding set of cache A or an empty block in cache B, we fill it. If not, for each set of cache A that has at least one block in cache B, we swap blocks if necessary, to make sure that among all blocks in caches A and B that are associated with each set, the block whose next reference is farthest in the future is in cache B. Now among the blocks in cache B, the block that is next referenced farthest in future is replaced (regardless of the set it is associated with). This choice is however not optimal in all cases.

For the example reference pattern in Figure 2, consider a design with direct-mapped cache A of size 2 blocks (2 sets) and a one block cache B. The references shown in Figure 2 are block addresses. Capital letters map to set 0 and small letters map to set 1 of cache A. The figure shows the contents of set 0 and set1 in cache A and cache B after each memory access. Using *pseudo-opt* replacement policy we incur 7 misses, whereas the minimum possible number of misses is 6. For example, at time 4, if we

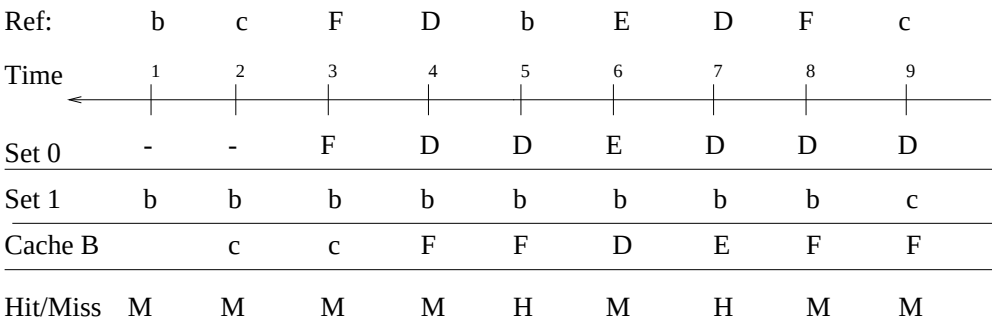

Figure 2: Misses using the pseudo-opt policy

replace block F (in set 0 of cache A) instead of block c (in cache B), we incur 6 misses instead of 7.

The above example shows that *pseudo-opt* policy is not optimal in all cases. However, if cache A is also a fully associative structure, caches A and B can be treated as a single fully associative cache. In this case the problem reduces to the case where Belady's design is optimal; hence our *pseudo-opt* replacement policy is also optimal. We are actively working on finding an optimal replacement policy for an M-way associative cache A.

For this initial experiment we use the *pseudo-opt* replacement policy. We use trace driven simulations to study the miss ratio of an 8KB direct mapped cache A plus a 1KB fully associative cache B. Table 1 also presents the results for a fully associative cache A.

Benchmark	**Miss Ratio**				
	DM(8KB)+FA(1KB)	**FA(8KB)+FA(1KB)**	**Victim**	**Assist**	**NTS**
compress	16.33%	16.29%	16.68%	16.68%	16.69%
gcc	3.99%	2.75%	6.7%	9.27%	9.2%
go	3.24%	2.18%	5.46%	7.77%	10.04%
ijpeg	2.08%	1.25%	3.61%	5.05%	5.73%

Table 1: *pseudo-opt* replacement - Miss Ratio

Although *pseudo-opt* is not an implementable policy, we use these results as a standard against which we measure the miss ratio of other cache designs. This helps us to understand the effect of replacement policies on miss ratio and indicates that better placement and replacement strategies are key to improving the performance of

these schemes. The NT-victim cache design is our first step in that direction.

3.3.2 NT-victim Cache Replacement Policies

Since the *pseudo-opt* replacement policy resulted in a 50% reduction in miss ratio, relative to Victim, Assist and NTS caches, it is apparent that replacement decisions in cache B are critical to improving performance. In order to understand how temporality can be exploited to guide our replacement decisions, we focus on the following three issues.

- *Cache space utilization*: Replacement decisions determine the population in the cache and therefore affect the utilization of the available cache space. Ideally a good replacement policy should *dynamically* partition the cache space for T and NT blocks, so that the available space in L1 is used effectively.
- *T-block priority*: Temporal reuse of a block is an indication of its usefulness. A replacement policy can give higher priority to T-blocks so as to retain them in favor of NT-blocks.
- *NT-block lifetime*: We quote from [15]:

  > "As the replacement algorithm chooses lines closer and closer to the most-recently used line, however, the risk becomes greater that the replacement algorithm will make a mistake and cast out a line that should be retained for a future hit."

  Therefore, in order to avoid replacing NT-blocks too soon, i.e., before they have a chance to stay in cache for a while longer and perhaps exhibit *temporal* reuse (which would give them more protection), our replacement policies offer various degrees of protection to NT-blocks so as to increase their lifetimes.

We now present three replacement policies for our NT-victim cache design. Our three replacement policies put varying emphasis on the above 3 issues and may therefore provide contrasting results for different benchmarks. Hence it is interesting to study the effect of each policy on the miss ratio. As we do not assume future knowledge of the program's memory reference trace, our policies depend on the history of the usage

patterns. Our goal is to improve performance by L1 caches by reducing the number of swaps without compromising the miss ratio improvement achieved by Victim caching.

- *Policy LRU*: In this policy we use the simple LRU replacement scheme. This policy does not use temporal information to make replacement decisions. Here *the least recently used block* in cache B is replaced.
- *Policy NT-LRU*: Here we exploit the temporal behavior of blocks in making the replacement decision. On a miss, we replace *the least recently used NT-block* in cache B. If there are no NT-blocks in B, we replace the least recently used T-block.
- *Policy NT-LRU-1/2*: This policy is a relaxed version of the NT-LRU policy. To offset the extremely preferential treatment given to T-blocks by NT-LRU, we pick a block for replacement only among the least recently used half of the blocks in cache B. Specifically, if a cache B has K blocks numbered 0 through K - 1, with block 0 being the most recently used block, we pick a block for replacement only among those with number $\geq \lceil (K/2) \rceil$. Among these blocks we choose the least recently used NT-block. If there are no NT-blocks among the least recently used half of the blocks in cache B, we pick the least recently used T-block for replacement.

NTS cache separates the reference stream into *temporal* and *non- temporal* and does not allow swaps. Cache blocks that are identified as *non-temporal* when they are replaced are allocated to cache B on subsequent requests. Restricting NT-blocks to cache B could lead to conflicts among NT-blocks for space in cache. This might affect performance by not utilizing all the available cache space. In order to achieve better space utilization of cache A and cache B, we allow NT and T blocks to reside in cache A and cache B.

Our first policy uses the simple LRU strategy as in Victim caching. This helps us to study the effect of temporal information on swaps. For instance, if the access pattern is a stride through an array, we observe that there is little or no temporal reuse. In such cases Victim caching and NT-victim caching with LRU policy may have the same miss ratio, but the NT-victim cache would incur fewer swaps.

NT-LRU policy clearly favors retaining temporal blocks. If the reference stream exhibits heavy temporal reuse (as in accessing an array repeatedly) and if the NT-blocks are few and far between, it is beneficial to give preferential treatment to T-blocks and retain them in favor of NT-blocks. However, for example, if the access pattern is a long sequence of *temporal* block accesses, followed by a long sequence of *non-temporal* block accesses, NT-LRU policy picks the NT-blocks for replacement even if the T-blocks may be least recently used and may have no future references. In such a case, the T-blocks can pollute the cache and, increasing the lifetime of NT-blocks proves to be more beneficial than replacing them. The NT-LRU-1/2 policy emphasizes this issue, tries to strike a more reasonable balance between simple LRU and favoring T-blocks and manages to retain some of the more recently used NT-blocks.

4 Experimental Setup

We now present details of the benchmarks used in this study, performance metrics analyzed, and the simulation environment.

4.1 Benchmarks

In numeric programs there are several known compiler techniques for optimizing data cache performance. However, integer programs often have irregular access patterns that are more difficult for the compiler to optimize. Therefore, we concentrate on the performance of integer benchmarks. In this section, we present the description of the benchmarks chosen for this study.

Simulating a cache that is much too small causes many capacity misses, which dominate the miss ratio and therefore obscures the effectiveness of a design in reducing conflict misses. On the other hand, simulating a cache that is much too large makes cache A sufficient by itself and therefore does not test the usefulness of cache B. Hence we study the miss ratio *vs.* cache size for each benchmark. In this study we use a direct-mapped L1 cache without cache B. This study gives us insights about the nature of the benchmarks and help us to choose a suitable size cache to simulate.

4.1.1 Description

All experiments were performed on the SPECint95 benchmarks (excluding vortex, m88ksim), which were compiled with gcc -O3. All benchmarks were simulated to completion. Table 2 describes the characteristics of the benchmarks used in this study.

Benchmark	Memory References	Data Set
099.compress	2,689,118	test.in
126.gcc	116,908,813	cccp.i
099.go	55,001,714	2stone9.in
132.ijpeg	174,309,186	specnum.ppm
130.li	735,318,824	test.lsp
134.perl	1,191,225,868	jumble.pl

Table 2: Benchmark Descriptions

4.1.2 Miss Ratio for Direct-Mapped Cache

In order to choose an appropriate size for cache A and cache B, we analyze the miss ratio *vs.* cache size for each benchmark using a direct-mapped L1 cache without any cache B. Starting from a cache size of 4KB we increase the size and study the miss ratio until there is little or no change in miss ratio with size.

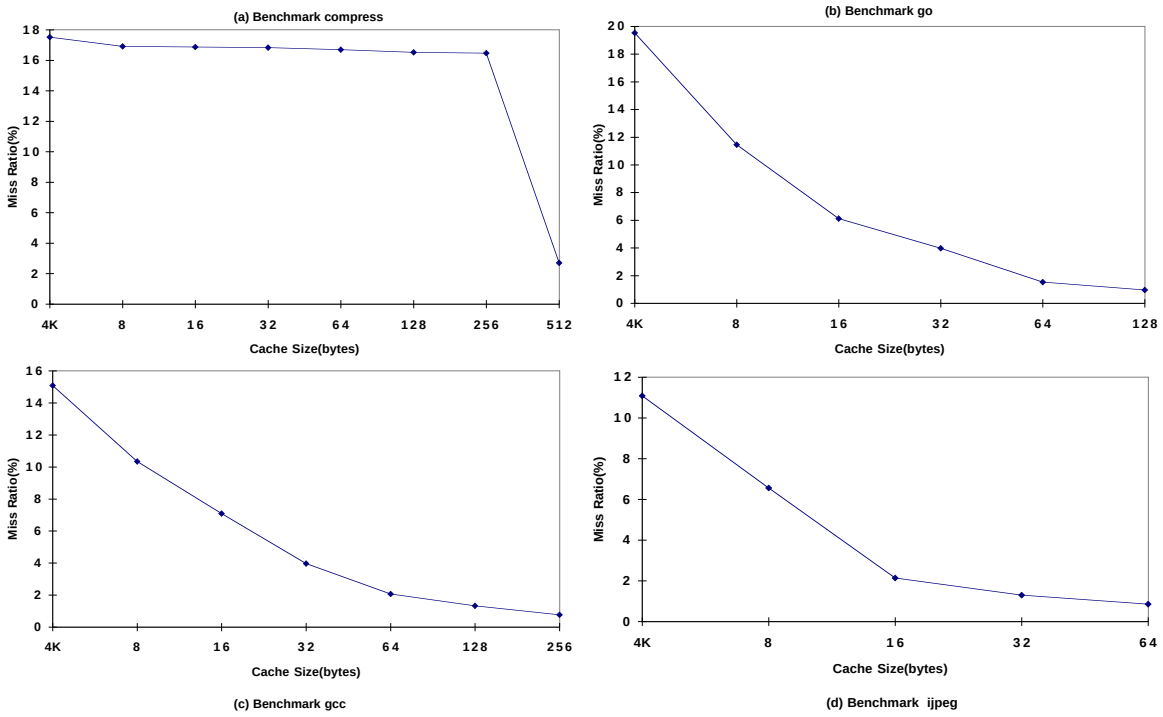

Figure 3: Miss Ratio *vs.* Cache Size

Compress sustains a high miss ratio as seen in Figure 3(a). Hence we conclude that for a direct-mapped cache A, we cannot get any performance improvement by adding

a small cache B in L1, unless the size of cache A is at least 256KB.

We observe from Figure 3(b), 3(c) and 3(d) for the benchmarks *go*, *gcc*, and *ijpeg*, the curves drop sharply at first and less steeply as the cache size increases. Most of the improvement in these graphs is obtained by the earlier doublings of the cache size. For each of these benchmarks, the knee in the curve is reached at different cache sizes. *Gcc* for instance, has the knee around 64KB, as does *go*, although *go* has a secondary knee at 16KB. *Ijpeg* has the smallest working set and the knee of the curve is around 16KB.

We also observe that the miss ratio drops to 1-2% around the knee of the curves. This shows that a design using only cache A (without any cache B) in L1 needs a large cache to achieve a substantial reduction in miss ratio. Therefore, we consider an alternative approach to doubling the size of L1 cache, namely, augmenting L1 with a small fully-associative cache B and achieving a similar reduction in miss ratio. For the benchmarks *go*, *gcc*, and *ijpeg*, the miss rate drops to nearly half its value when we increase the cache size from 8KB to 16KB. In addition, the average miss ratio for the benchmarks in this study is around 5% for a 16KB cache. Since increasing associativity generally decreases conflict misses, we chose to couple an 8KB direct-mapped cache A with a small (1KB) fully associative cache B to relieve conflict pressure.

4.2 Performance Metrics

The goal of this study is to develop a novel microarchitecture for an L1 cache design, by exploring the design space between Victim cache and NTS cache. Our design selectively uses temporality information in making replacement and swap decisions. We have observed that victim caching is very effective in reducing conflict misses. The goal of an NT-victim cache design is to reduce cache pollution and the number of swaps, without compromising the miss rate achieved by Victim cache. For all cache designs that we study, we assume that a hit in cache B incurs no additional cycles over a hit in cache A and that all the designs have the same miss penalty for fetching a block from the next level of the hierarchy.

In this work we concentrate on two major performance metrics:

- the miss ratio
- the number of swaps.

In order to study the importance of swaps, we analyzed the miss ratio without swaps for both Victim cache and NT-victim cache. We found that the miss ratio *increases* without swaps. This suggests that swaps do help in retaining more useful cache blocks in cache A, and in turn help in picking a block for replacement from among the less useful blocks. Although NT-victim caches do not incur additional delay for a hit in cache B, we still need to swap blocks to get better performance. This motivates us to design a cache which reduces the number of swaps compared to a Victim cache, without compromising the miss ratio.

4.3 Simulator

We use trace-driven simulations to evaluate the performance of NT-victim cache. A Memory reference trace of each benchmark is collected using ATOM [13]. We simulate Victim, NT-victim, Assist, and NTS caches. As a base case, we also simulate a direct-mapped L1 cache, without any cache B. In the case of NT-victim caches we simulate all the three replacement policies, namely, NT-LRU, LRU and NT-LRU-1/2.

Due to storage constraints, we could not collect very large ($>$ 2GB) traces. Therefore, we use execution-driven simulations for those benchmarks. We still use ATOM to instrument the executables of the benchmarks, but when each load/store instruction of the instrumented program is executed, it calls simulators for Victim, NT-victim and base case direct-mapped caches.

In this study we focus on miss ratio and the number of swaps; we plan to analyze latency effects in future work. Hence we have simulated our design using a timing simulator - *mlcache* [16]. This simulator models latencies due to bus width considerations, bus contention, trailing-edge effects, port limitations and the number of outstanding accesses that the cache allows before it blocks. *mlcache* provides a library of routines that a user may choose from to determine what actions will take place in the cache for a given event. These routines are useful for simulating different cache designs and comparing results.

For each design we simulate an 8KB direct-mapped cache A and 1KB fully-associative cache B. Cache block size is uniformly fixed at 32 bytes. Choice of cache size is based on our study of miss ratio of the benchmarks *vs.* cache size for a direct-mapped L1 cache without cache B. Results of this study indicate that the average miss ratio for our set of benchmarks, for a cache of size 8KB is only 9%. The miss rate halves when we double the cache size to 16KB. Our goal here is to achieve the performance of a 16KB direct-mapped cache by using an 8KB direct-mapped cache A and a 1KB fully-associative cache B. All cache configurations are assumed to be copy-back and write-allocate. Except for copy-back effects, there are no special considerations to distinguish LOAD and STORE references.

5 Results

5.1 Miss ratio

The cache miss ratios for the base case (direct-mapped cache without cache B), Victim cache, NTS cache, Assist cache and NT-victim cache are shown in Figure 4. For each benchmark, we also present the miss ratio for each cache design in Table 3. For NT-victim cache we present the miss ratio for each of the replacement policies described in section 3.3.2.

Compress has a high miss ratio for the configurations simulated. We have examined the accessing behavior of this application in detail. We simulated this application for various L1 cache sizes up to 512K and found little or no change in miss ratio until after 256K. For a cache size of 512K the miss ratio dropped to 3% as shown in Figure 3. As our design focus is on performance improvement of direct-mapped caches, we conclude that the working set of this benchmark is too large to fit into direct-mapped caches of size less than 512K. This agrees with the conclusion in [6]. Johnson and Hwu also observe that many of the memory accesses in *compress* are to its hash tables, *htab* and *codetab*. Due to large hash table sizes and the fact that the hash table accesses have little temporal or spatial locality, there is very little reuse in L1 cache. Figure 5 shows that even for the Victim cache, 97% of the replacements are NT-blocks. As Victim caches merely pick the LRU block for replacement, irrespective of whether it is a T

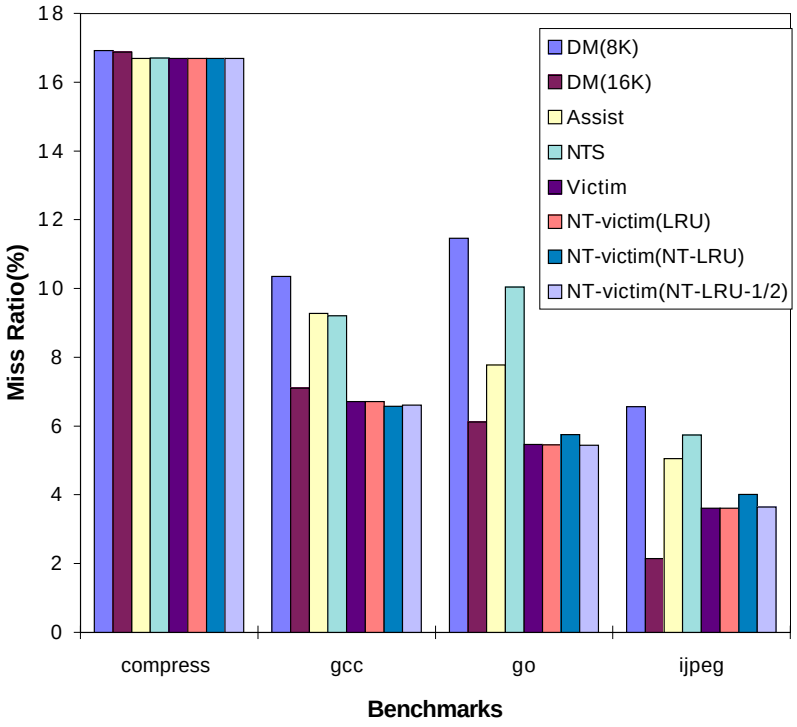

Figure 4: Miss Ratio

or an NT-block, this implies that 97% NT-blocks are also LRU blocks. Hence there is very little temporal reuse exhibited in the code for this cache size. We also observe that doubling the cache size (DM 16K) has negligible effect on the miss ratio.

All the other benchmarks in our suite exhibit similar trends in their results. For these benchmarks, our results shown in Table 3 and Figure 4 confirm that adding a small fully-associative buffer to L1 generally (i.e., except for *ijpeg*) eliminates more conflict misses than simply doubling the size of cache A. For the benchmarks *gcc*, *go*, *perl*, and *li* we see that a direct-mapped 16KB cache still does not perform as well as the Victim or NT-Victim cache, which has only an 8KB direct-mapped cache A with a 1KB fully associative cache B. However, if the working set of the benchmark for a direct-mapped cache is small, like the one for the benchmark *ijpeg*, then doubling the cache size might reduce the miss ratio more than our NT-victim design alternative.

Recall that in section 3.3.2, we identified three issues as key to improving the performance of our design. We now re-visit these issues in light of the observed results.

Cache Configuration	Miss Ratio for Benchmarks					
	compress	*gcc*	*go*	*ijpeg*	*li*	*perl*
DM(8KB)	16.91%	10.34%	11.46%	6.56%	-	-
DM(16KB)	16.87%	7.09%	6.11%	2.14%	5.16%	3.55%
Assist	16.68%	9.27%	7.77%	5.05%	-	-
NTS	16.69%	9.2%	10.04%	5.73%	-	-
Victim	16.68%	6.7%	5.46%	3.61%	3.93%	2.36%
NT-Victim(LRU)	16.68%	6.7%	5.45%	3.6%	3.93%	2.36%
NT-Victim(NT-LRU)	16.68%	6.57%	5.75%	4.01%	3.94%	2.45%
NT-Victim(NT-LRU-1/2)	16.68%	6.6%	5.44%	3.64%	3.86%	2.34%

Table 3: Miss Ratio

- *Cache space utilization*: Our results confirm that allowing both NT and T blocks to reside in cache A and in cache B achieves a better utilization of the cache space than allowing only NT-blocks to reside in cache B. Therefore, NT-victim cache performs better than NTS cache for all these benchmarks. For example, for the benchmark *gcc*, Table 3 shows that on average Victim caches and NT-Victim caches achieve a 33% reduction in miss rate relative to NTS cache. Similarly, for the benchmark *go*, Assist cache has a lower miss ratio (7.77%) than the NTS cache (10.04%). Since NTS cache restricts NT-blocks to cache B, its high miss rate could be due to conflicts among *temporal* blocks and among *non-temporal* blocks. Allowing NT and T blocks to reside in both caches seems to relieve this conflict and achieves better cache space utilization.

- *T-block priority*: NT-LRU replacement policy favors retention of T-blocks over NT-blocks. In most cases, this scheme seems to unfairly penalize NT-blocks. Table 3 shows that for the benchmarks *go*, *ijpeg*, *perl* and *li*, NT-victim cache using NT-LRU policy has a higher miss ratio than Victim cache or NT-victim cache using LRU or NT-LRU-1/2 replacement policy. However, for a benchmark with irregular memory access patterns, favoring retention of T-blocks over NT-blocks could prove beneficial, as shown by our results on *gcc*. The benchmark *gcc* has the largest number of dynamic memory accesses; it has 124 *malloc* calls in the source code, which leads to irregular memory accesses. Table 3 shows that for *gcc*, NT-victim caches using NT-LRU replacement policy gives the lowest miss

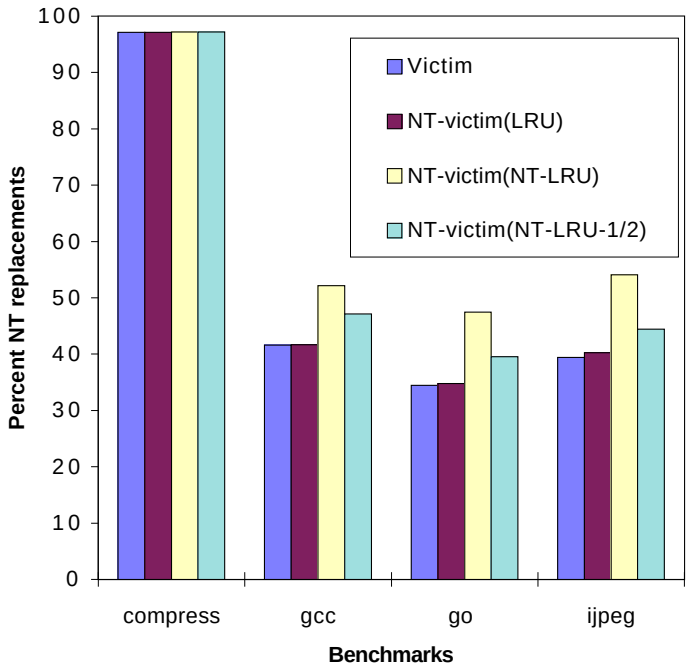

Figure 5: Percent NT replacements

ratio (6.57%). In addition, Figure 5 shows that for this benchmark, under the NT-LRU policy, 52% of the replacements are NT-blocks. Giving higher priority to T-blocks is beneficial in this case.

- *NT-block lifetime*: NT-LRU-1/2 replacement policy relaxes the special treatment given to T-blocks and gives NT-blocks a better chance to continue residing in the cache. Our results in Table 3 show that in most cases NT-victim cache with the NT-LRU-1/2 policy performs better than Victim cache. This implies that temporality based replacement decisions are important to improve performance. However, we also observe that this improvement is not very high. For the benchmarks *go*, *li*, and *perl*, NT-victim cache using NT-LRU-1/2 policy gives the lowest miss ratio among all the cache configurations. Figure 5 also confirms that the percentage of NT-replacements for these benchmarks is lower when using the NT-LRU-1/2 policy rather than the NT-LRU policy. This shows that giving NT-blocks a better chance of staying in cache, makes them more likely to exhibit temporal reuse and become T-blocks, which in turn results in improved perfor-

mance. These results also indicate that NT-LRU policy overpenalizes NT-blocks and hence increases miss ratio. On the other hand, the NT-LRU-1/2 policy uses temporal information to make replacement decisions, yet protects NT-blocks from being replaced too soon, and thereby improves the miss ratio.

The above results show that NT-Victim caches can indeed have miss ratios comparable to Victim caches. We now proceed to compare the number of swaps in each of these designs.

5.2 Swaps Between Cache A and Cache B

In a Victim cache a hit in cache B incurs an additional penalty cycle; and therefore, when a block is hit in cache B, it is swapped with the LRU block of that set in cache A. In our current model there is no penalty for a hit in cache B. As in the design of Assist cache [8], we place caches A and B in parallel, in the same level of the memory hierarchy.

In order to understand the effect of swaps on the miss ratio we study the miss ratio without swaps for both Victim and NT-victim caches. In this experiment, when a block is a hit in cache B, it continues to remain in cache B. The results of this study are summarized in Table 4. Comparing these results with our results in Table 3, we see that the miss ratio *increases* when swaps are not allowed. The larger size of cache A helps to retain useful blocks for a longer time. We therefore conclude that swaps are indeed useful in improving the performance.

Cache Configuration	**Miss Ratio for Benchmarks**					
	compress	*gcc*	*go*	*ijpeg*	*li*	*perl*
Victim	16.69%	7.03%	5.85%	3.69%	3.69%	5.22%
NT-victim(LRU)	16.69%	7.03%	5.85%	3.69%	3.69%	5.22%
NT-victim(NT-LRU)	16.68%	6.92%	5.82%	4.03%	3.73%	5.31%
NT-victim(NT-LRU-1/2)	16.68%	6.96%	5.78%	3.72%	3.68%	5.19%

Table 4: Miss Ratio without Swaps

We now address the goal of reducing in the number of swaps without compromising the miss ratio. Unlike Victim caches, in NT-Victim caches we do *selective* swapping

on a hit in cache B, by swapping a block in cache B with a block in cache A *only* if it exhibits temporal reuse. This selective swapping does reduce the number of swaps, relative to Victim caches, and does maintain a comparable miss ratio.

Cache Configuration	**Swaps for Benchmarks**					
	compress	*gcc*	*go*	*ijpeg*	*li*	*perl*
Victim	6,265	4,250,374	3,300,142	5,135,926	23,760,351	41,402,635
NT-victim(LRU)	5,641	4,096,771	3,094,852	4,226,530	22,531,514	40,226,357
NT-victim(NT-LRU)	5,292	4,335,089	3,074,227	4,235,153	22,686,224	40,765,365
NT-victim(NT-LRU-1/2)	5,370	4,231,092	3,153,037	4,330,588	22,750,042	41,105,485

Table 5: Swaps between Cache A and Cache B

Table 5 shows the number of swaps that occur for NT-victim and Victim caches. The number of swaps for NT-victim cache is consistently less than for Victim cache. We now return to the 3 issues that we identified in section 3.3.2 and discuss how controlling the number of swaps relates to these issues.

- *Cache space utilization*: We observe from Table 5 that *selective* swap policy reduces the number of swaps, helps us retain more useful blocks in cache A and further improves cache space utilization.

- *T-block priority*: Our result on miss ratio in Table 3 shows that NT-LRU replacement policy penalizes NT-blocks and does not give those that should be T-blocks, a chance to remain in cache long enough to exhibit temporal reuse. However, we see from Table 5 that the NT-LRU replacement policy incurs the smallest number of swaps for the benchmarks *compress* and *go*. This indicates that although replacing NT-blocks more frequently reduces their chance to exhibit *temporal* reuse, it does result in fewer swaps. On the other hand, the LRU and NT-LRU-1/2 policies do provide greater chance for NT-blocks to become temporal and hence they do incur more swaps.

  Contrary to the above general trend, results for *gcc* in Table 5 show that the number of swaps for Victim cache is less than for NT-victim cache with the NT-LRU replacement policy. This exception to the above general pattern can be explained as follows: Results in Table 3 indicate that for *gcc*, NT-victim cache

with the NT-LRU policy has the lowest miss ratio (6.57%) and NTS cache shows a high miss ratio (9.2%). Since NT-victim cache with the NT-LRU replacement policy retains T-blocks in cache B, it eases the conflicts among T-blocks for space in cache A and therefore improves the miss ratio. However, retaining T-blocks results in *temporal* hits in cache B and hence more swaps.

- *NT-block lifetime*: Protecting NT-blocks from being replaced too soon, gives them a greater chance to exhibit temporal reuse and become T-blocks. As a result, increasing the lifetime of an NT-block should generally increase the number of swaps. This is confirmed by our results in Table 5. However, our results from the previous section indicate that increasing the lifetime of NT-blocks also improves miss ratio.

Our results in this section confirm that using temporal information in swap decisions can indeed improve performance by reducing the number of swaps without compromising the miss ratio.

6 Conclusions and Future Work

As instruction issue width increases, more demand will be placed on the data cache. In this work we described several previously proposed cache designs in which a conventional L1 cache is augmented by a fully associative buffer to improve performance. After studying the performance improvement achieved by each of these designs, we focussed on two of the best designs: Victim and NTS cache.

Victim cache reduces conflict misses and gives the lowest overall miss ratio. However, Victim caches do incur a large number of swaps between the main cache and the victim buffer. NTS cache exploits the *temporal* locality of blocks to improve the performance of the L1 cache and does not swap blocks between the main cache and the NTS buffer. This might affect performance by not utilizing all the available cache space effectively.

Taking the lead from these two designs, we design a new cache, which we call NT-victim cache. In this scheme we have a small fully associative buffer similar to the victim buffer. The goal of the design is to reduce conflict misses and have a miss ratio

comparable to Victim caches, while also reducing the number of swaps by utilizing information about whether the blocks have exhibited *temporal* behavior.

Our results show that even though no additional penalty is incurred in accessing a block from cache B, swapping blocks does help retain more useful blocks in cache A. However, swaps do incur additional cycles. Hence a reduction in the number of swaps is important, provided that the low miss ratio is maintained. Our results confirm that *temporal* behavior is quite useful in determining whether to retain a block in cache B or swap it with a block in cache A. Selectively swapping blocks based on their temporal reuse does in fact reduce the number of swaps and does not increase the miss ratio. However, our attempts to exploit NT/T information in our replacement policy did not have a large effect on performance. The three key issues that we identified, namely,

- *Cache space utilization*
- *T-block priority*
- *NT-block lifetime*

are important in improving the performance of our design. Our results show that allowing NT and T blocks to reside in both caches A and B does improve performance and provides better cache space utilization relative to the NTS cache. The NT-LRU replacement policy is very aggressive in retaining T-blocks; but overpenalizes NT-blocks. Our results for the NT-LRU-1/2 policy confirms that protecting NT-blocks from being replaced too soon and thereby increasing their lifetimes, improves performance. In future work we plan to explore a wider space of cache configurations, policies and applications.

Our results from the pseudo-optimal replacement policy suggest that for our cache model, there is a big gap between currently achieved performance and the maximum achievable performance. We are interested in finding an optimal replacement policy for our cache model and analyzing its behavior in detail in order to derive better implementable designs. A deeper understanding of the optimal policy may suggest ways to further improve the placement and replacement decisions of our cache model.

7 Acknowledgments

This work is the result of many fruitful discussions I have had with Prof. Ed Davidson. Here is a special thanks to him. Many thanks are due to Jude Rivers and Ed Tam for their insights and encouragement in this project and their patience in answering my many questions.

References

[1] A.Agarwal, S.D.Pudar, Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct Mapped Caches, *Proceedings of the 20th Int'l Symposium on Computer Architecture, May 1993 pp 179-190.*

[2] L.A. Belady, A Study of Replacement Algorithms for a Virtual Storage Computer, *IBM Systems Journal, Vol. 5, 1966, pp.78-101.*

[3] James E. Bennett, Michael J. Flynn, Prediction Caches for Superscalar Processors *To be published in Proceedings of the 30th Annual Int'l Symposium on Microarchitecture November 1997.*

[4] C-H.Chi, H. Dietz, Improving Cache Performance by Selective Cache Bypass *Proceedings of the 22nd Annual Hawaii Int'l Conference on System Science,Jan 1989.*

[5] M.Hill, A Case for Direct-Mapped Caches *IEEE Computer, Vol.21, No.12, Dec 1988, pp 25-40.*

[6] Teresa Johnson , Wen-mei W.Hwu, Run-time Adaptive Cache Hierarchy Management via Reference Analysis *Proceedings of the 24th Int'l Symposium on Computer Architecture, June 1997 pp 315-326.*

[7] N.P.Jouppi, Improving Direct-Mapped Cache Performance by the Addition of a Small Fully-Associative Cache and Prefetch Buffers, *Proceedings of the 17th Int'l Symposium on Computer Architecture, May 1990 pp 364-373.*

[8] G.Kurpanek, K.Chan, J.Zheng, E.DeLano, W.Bryg, PA7200: A PA-RISC Processor With Integrated High Performance MP Bus Interface, *COMPCON Digest of Papers, Feb 1994, pp. 375-382.*

[9] Yale N. Patt, Sanjay J Patel, Marius Evers, Daniel H. Friendly Jared Stark, One Billion Transistors, One Uniprocessor, One Chip *IEEE Computer, Vol.30, No.9, Sept 1997, pp 51-57.*

[10] Jude A. Rivers, Edward S. Davidson, Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design, *Proceedings of the 1996 International Conference on Parallel Processing, Vol I, August 1996, pp 151-162.*

[11] Jude A. Rivers, Edward S. Tam, Edward S. Davidson, On Effective Data Supply for Multi-Issue Processors, *Proceedings of the 1997 IEEE International Conference on Computer Design, October 1997, pp 519-528.*

[12] Alan J. Smith, Cache Memories *Computing Surveys, Sept. 1982, pp 473-530.*

[13] A.Srivastava, A.Eustace, ATOM: A System for Building Customized Program Analysis Tools, *Proceedings of ACM SIGPLAN Notices Conference on Programming Language Implementations, June 1994, pp 196-205.*

[14] Dimitrios Stiliadis, Anujan Varma, Selective Victim Caching: A Method to Improve the Performance of Direct Mapped Caches, *IEEE Transactions On Computers, Vol 46, No. 5, May 1997 pp 603-610.*

[15] Harold S. Stone, *High-Performance Computer Architecture, Addison-Wesley Publishing Company, Inc., 1993, pp 74-75.*

[16] Edward S Tam, Jude A. Rivers, Edward S Davidson, Flexible Timing Simulation of Multiple-Cache Configurations, *CSE Tech Report, University of Michigan.*

[17] O.Temam, N.Drach, Software Assistance for Data Caches *Proceedings of 1st International Symposium on High Performance Computer Architecture, Jan 1995, pp 154-163.*

[18] Gary Tyson, Matthew Farrens, John Matthews, Andrew R. Pleszkun, Managing Data Caches Using Selective Cache Line Replacement *Int'l Journal of Parallel Programming, Vol. 25, No. 3, 1997 pp 213-242.*