Design and Performance Evaluation of a Cache Assist to implement Selective Caching

L. John
Electrical and Computer Engineering Department
University of Texas at Austin
Austin, TX 78712
ljohn@ece.utexas.edu
kurian@csee.usf.edu

A. Subramanian
Computer Science and Engineering Department
University of South Florida
Tampa, FL 33620

Abstract

Efficient instruction and data caches are extremely important for achieving good performance from modern high performance processors. Conventional cache architectures exploit locality, but do so rather blindly. By forcing all references through a single structure, the cache's effectiveness on many references is reduced. This paper presents a selective caching scheme for improving cache performance, implemented using a cache assist namely the annex cache. Except for filling a main cache at cold start, all entries come to the cache via the annex cache. A block from the annex cache gets swapped with a main cache block only if it has been referenced twice after the conflicting main cache block was referenced. Essentially, low usage items are not allowed to create conflict misses in the main cache. Items referenced only rarely will be excluded from the main cache, eliminating several conflict misses and swaps. The basic premise is that an item deserves to be in the main cache only if it can prove its right to exist in the main cache by demonstrating locality. The annex cache has some of the features of a victim cache, but the processor can access annex cache entries directly, i. e. annex cache entries can bypass the main cache. Extensive simulation studies for annex and victim caches using a variety of SPEC programs are presented in the paper. Annex caches were observed to be significantly better than conventional caches and comparable to victim caches, while providing hardware advantages in comparison to victim caches.

Keywords: Cache exclusion, cache bypassing, selective caching, cache replacement, cache assists, victim caches.

** This work was supported in part by a CAREER Award from the National Science Foundation (Grant CCR-0624378) and a grant from Oak Ridge Associated Universities.*

1 Introduction

Processor speeds have increased dramatically in the past few years, but DRAM speeds have not kept pace resulting in a large gap between CPU and DRAM speeds. Technological trends suggest that this speed gap is on the increase. These factors have led to extensive research on architectural techniques to reduce or hide memory latency, including cache hierarchies, lockup-free caches, load hoisting, hardware and software prefetching, etc.

While designing cache systems, there are several tradeoffs involving mapping schemes, replacement strategies, fetch policies etc. Direct-mapped caches perform better than set-associative caches when the access time costs for hits are considered [11]. Direct-mapping is the only cache configuration where the critical path is merely the time required to access a RAM. But they have more conflict misses due to their lack of associativity.

To deal with the higher miss rates in direct-mapped caches, designers and architects have developed various hybrid caches that combine fast access with greater associativity by providing a fast direct-mapped primary section together with a secondary section with higher associativity. Some examples of such schemes to improve performance of direct mapped caches are victim caches [11], hash-rehash caches [2], column-associative caches [3], MRU caches [6] and half-and-half caches [26].

In this paper, we propose another hybrid cache, the annex cache. Except for filling a main cache at cold start, all entries come to the cache via the annex cache. An entry from the annex cache gets swapped with a main cache entry only if it has been referenced twice after the conflicting main cache entry was referenced. Essentially, low usage items are not allowed to create conflict misses in the main cache. Items referenced only once or twice will be excluded from the main cache, eliminating several conflict and capacity misses. The basic premise is that an item deserves to be in the cache only if it can prove its right to exist in the cache.

The original motivation while designing the annex cache was to implement selective caching. One flaw associated with the conventional caching mechanism is that it exploits locality in a blind manner. Since all references are forced through the cache, every miss will result in a new block of information entering the cache. This new block of information could be a piece of rarely used data, but it may replace a piece of heavily used data and result in additional misses. Thus high usage instructions with a long live-range may be knocked off by infrequent instructions, resulting in increased miss rates and lower cache performance. It is our hypothesis that the cache miss problem can be alleviated by excluding infrequent instructions from the cache. The annex cache implements bypassing or cache exclusion for infrequent items.

1.1 Related research

The original motivation while designing the annex cache was to implement selective caching. However, once we designed the mechanism, it looked very similar to victim caches and half and half caches. Hence we will describe related research in both avenues here.

McFarling [19] proposed a mechanism to dynamically decide whether an instruction causes conflicts and should be excluded from the cache. The decision whether an instruction should be replaced when a competing instruction is needed or whether the competing instruction should just be passed on to the processor bypassing the cache is made by a special finite state machine (FSM) in conjunction with two or more state bits associated with each cache block. Gonzalez et al [9] presented a dual data cache with independent parts for managing spatial and temporal locality. The dual data cache make use of a 'locality prediction table' to delay caching something until a benefit in terms of locality can be predicted.

Cache bypassing has also been studied in the past by Chi and Dietz [8], Abraham et. al. [1], etc. Chi and Dietz [8] showed that bypassing the cache can avoid cache pollution and improve performance for data loads and stores. Abraham et. al. [1] showed that *labeled* load/store instructions can be used to optimize cache behavior. Their HPL PlayDoh architecture provides explicit control of the memory hierarchy and supports prefetching of data to any level in the hierarchy. Each load instruction is tagged with a specifier corresponding to the cache or main memory.

In the past, there have been several schemes to improve performance of direct mapped caches, for example, victim caches [11], hash-rehash caches [2], column-associative caches [3], MRU caches [6] and half-and-half caches [26]. Theobald et al [26] introduced half and half caches and also provided a concise summary of the other schemes. Bershad et. al. [5] introduced another cache assist, the Cache Miss LookAside Buffer to record a history of cache misses and dynamically remap pages by using a software policy in the operating system's virtual memory management system. Temam and Drach [25] presented the concept of large *virtual cache lines* to exploit spatial locality and a *bounce-back cache* to exploit more temporal locality.

1.2 Contributions

The proposed scheme is most similar to McFarling's dynamic cache exclusion scheme [19] and Jouppi's victim caches [11] or Theobald and Gao's half-and-half caches [26]. However, there are notable differences. In the victim cache, every hit in the victim cache is accompanied by a swap between the main cache and the victim cache. In the annex cache, swaps are performed only for references that have demonstrated locality. Other references that hit in the annex cache are simply bypassed to the processor.

In most studies on victim cache and half-and-half caches, it is assumed that the swap happens in a single cycle even if the block size is 32 or 64 bytes. In order for the swap to happen in a single cycle, a wide datapath between the main and victim cache would be required. The annex cache we propose makes fewer swaps because of the bypassing algorithm. Hence the scheme performs well even if the swap hardware is not very sophisticated.

McFarling's finite-state-machine [19] to implement cache bypassing uses two bits to capture the reference pattern, the sticky bit and the hit-last bit. The hit-last-bit has to be updated even when the block is not in the primary cache. The scheme we propose employs only one extra bit that is associated with every block in the cache. Recently there has been studies suggesting the use of two-level exclusive on-chip caches [12]. Our algorithm works very well with on-chip multi-layer caches, makes more efficient use of the on-chip cache space, reduces the hardware complexity of the swapping hardware, and reduces the overhead associated with cache exclusion compared to McFarling's scheme. McFarling [19] investigated the effectiveness of cache bypassing but did not compare it with victim caching. According to that paper, *victim caches work well for data references where the number of conflicting items may be small. For instruction references, there are usually many more conflicting items than a victim cache can hold. This is where dynamic exclusion is most effective.* But no quantitative results comparing victim caches with selective caching was presented in [19] or other research on selective caching [8] [1]. We perform a quantitative comparison.

In section 2, the proposed annex caching scheme is described. In section 3, a quantitative analysis of the performance of the selective caching scheme based on trace-driven simulation is presented. Section 4 presents summary and concluding remarks.

2 The Annex Cache

In this section, we describe the proposed cache assist, Annex Cache. Just like in victim and half and half caches, the main cache is direct mapped and the assist cache has higher associativity. In general two-level caching, the second level is typically off-chip, but in the annex cache, both levels are on-chip as in Theobald et al's half and half caches [26] and Jouppi and Wilton's split-level caches [12].

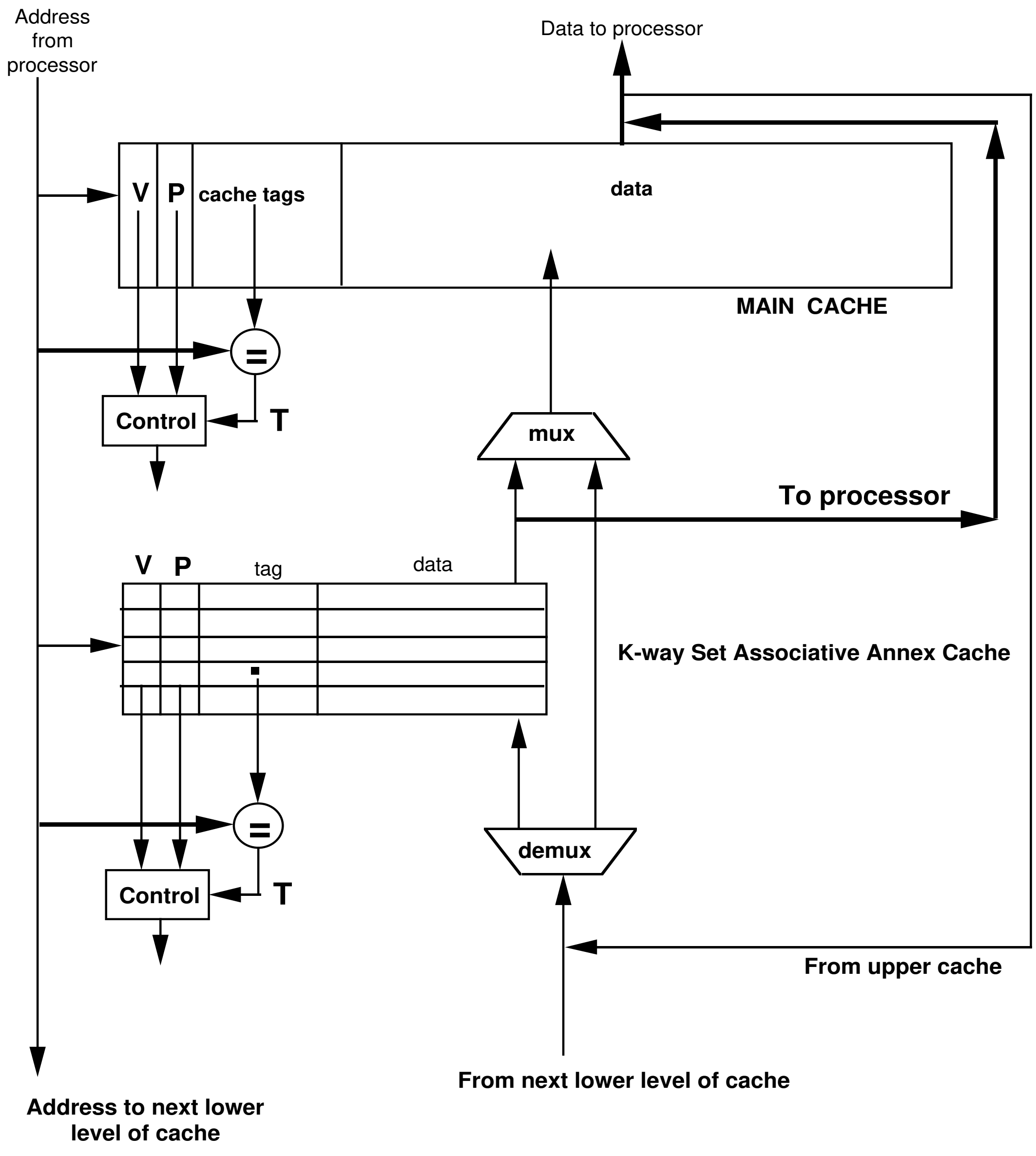

Figure 1: Annex Cache Organization

In victim cache, the cache assist is meant to hold the items that would otherwise be displaced from the cache. But if the conflicting entries in the main cache and the assist cache are referenced alternately, each access would result in a swap between the main and the assist cache. The annex cache eliminates such continuous swapping. One of the entries will be accessed from the main cache and the other entry will be directly accessed from the assist cache. The assumption is that a mechanism exists for the assist cache entries to be fed directly to the processor bypassing the main cache. We name the assist cache with the above properties as the annex cache.

The annex cache may be viewed as a "qualifying station" or as an "entry-level cache", where the references have to prove themselves to be eligible to enter the main cache. It may also be viewed as a "double-hit victim cache", where the swapping of the main and assist cache occurs only at the second reference to an assist-cache entry with no intervening reference to the conflicting main cache entry.

The proposed scheme works in the following manner. Each time the upper cache is probed, the annex cache is probed as well. If a miss occurs in the upper cache, but the address hits in the annex cache, then the main cache can be reloaded in the next cycle from the annex cache (if desired). This replaces a long off-chip miss penalty with a short 1-cycle off-chip miss. This arrangement satisfies the requirement that the critical path is not worsened, since the annex cache itself is not in the normal critical path of processor execution.

Every cache block has a priority bit in addition to the valid bit. If there is a hit, the data is read from the main cache and the priority bit is set to 1. If there is a miss, the annex cache is searched. If there is a hit in the annex cache, the annex and main cache entries are swapped or the data is read directly from the annex cache depending on the priority bits of the conflicting elements. The details of the algorithm are presented in Fig. 2. If an instruction is kept in the main cache despite a conflict, it will be replaced the next time a conflicting instruction is executed unless the original instruction is executed first. The algorithm basically introduces some inertia in replacing an item from the main cache. An item gets promoted to the main cache only if it has proven its right to exist there by showing high usage from the annex.

One aspect of annex caches is that they violate inclusion properties [4] in cache hierarchies. As in victim caching, no item appears both in the main cache and the annex cache.

2.1 Algorithm:

1. In the first cycle, the cache reads the tags and data of the relevant block in the primary cache and in the annex cache. The data from the primary cache is latched into the output buffer becoming available at the end of the first cycle.

2. The tag from the main cache and the K tags from the annex (annex is K-way set associative) are compared against the appropriate bits of the address.

3. If the tag from the main cache matches the address, then a main cache hit occurs (path 8 in Fig. 2). The priority of the main cache entry is updated to be high. The search in the annex cache is immediately cancelled so that it can be used again in the next cycle. If the tag from the main cache does not match, the annex search has to be continued.

4. If there was no main cache hit and if one of the tags from the annex match, an annex-cache hit occurs. The data is read directly from the annex if the conflicting main cache entry has a high priority (path 7 in Fig. 2) or if the annex cache entry has a low priority (path 4). Otherwise, the

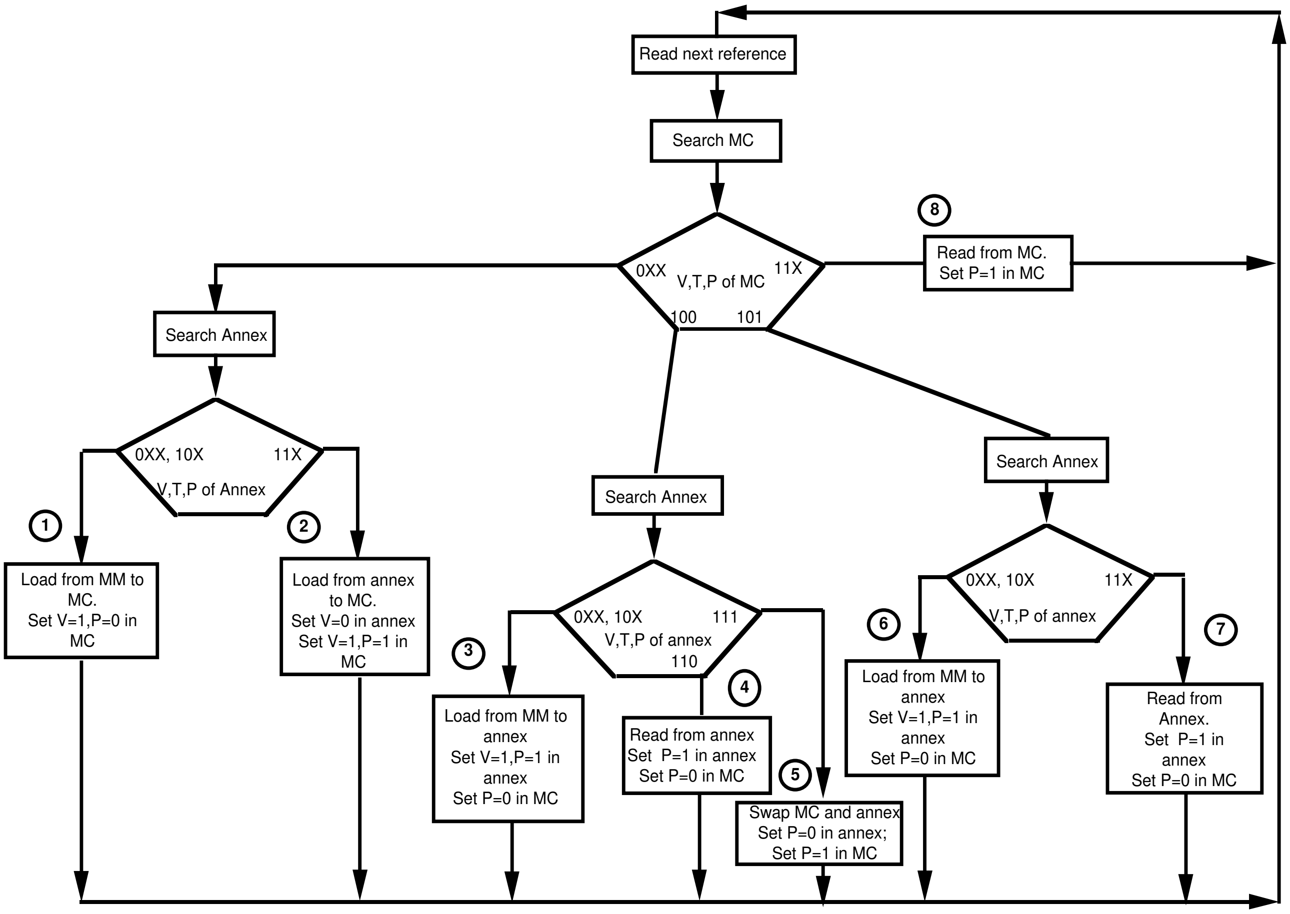

Figure 2: Algorithm for the Annex Cache

data is loaded into the main cache (path 2) or swapped with the conflicting main cache entry (path 5).

5. If all tag comparators mismatch, the data is in the main memory. When the data returns from the main memory, it is placed in the main cache if the corresponding main cache block has no valid data (path 1);otherwise it is placed in the annex cache (paths 3 and 6).

In the different paths, the different priority bits are set as shown in Fig. 2.

2.2 Tuning of the algorithm:

(i) At cold start should items come through the annex or directly?

In our algorithm, we investigated the effect of bringing in items to the main cache always through the annex cache. As expected, it was not good because the main cache is left empty and is underutilized.

(ii) When an entry reaches the main cache at cold start, should its priority be low or high?

Since the item reached the main cache without establishing locality, we assigned low priority and on experimentation with both priorities, our choice was seen to be better.

(iii) How soon should an item in the annex be allowed to enter the main cache?

If the inertia is high, there will be too many initial misses and performance is expected to be low. This was confirmed experimentally also. According to the current algorithm, in (AB^3), reference B will get cached at the second access to B. This choice results in several swaps in the sequence $(ABC)^{10}$, By increasing the inertia, we could avoid any swaps between the main cache and the annex cache in $(ABC)^{10}$, but then there are increased misses in sequences such as (AB^3). Even in the current algorithm, there will be only half the number of swaps in annex compared to victim or half and half cache for the sequence $(ABC)^{10}$. When an item reaches the annex the first time, currently we set its priority to be high because it was just referenced. Initially we used to set it to be low, which increases the time to reach main cache if it has strong locality, but higher priority was seen to be better.

2.3 Illustration for Simple Sequences

Let us examine how effective this scheme will be, for some simple reference patterns also used by McFarling [19] to illustrate his dynamic cache exclusion scheme.

2.3.1 Conflict within Loops

Consider a case where two references A and B within a single loop map to the same location in the cache. If the loop is executed 10 times, the memory access pattern may be represented as $(AB)^{10}$, where the superscript denotes the frequency of usage of the particular instruction. If we allow both instructions to enter the cache, each instruction will knock each other out of the cache and neither hits. Hence the behavior of a conventional cache is

$$(A_m B_m)^{10}$$

where A_m denotes that reference A is a miss. A subscript h would indicate a hit. In this case, the miss rate of a conventional cache is

$$M_{conv} = 100\%.$$

Now let us consider the behavior of a victim cache to this sequence. In the conventional cache, A would be replaced by B and every reference would be a miss. In the victim cache, the when A gets replaced by B, A would be put into the victim cache; hence after the initial miss to A and B, there are no further misses. The main cache misses will be filled from the victim cache. Let us use A_{h-v} to represent that A is a hit in the victim cache and A_{h-m} to represent that A is a hit in the main cache. A miss in both the victim as well as main caches will be represented as A_m. The behavior of a victim cache is

$$A_m B_m (A_{h-v} B_{h-v})^9$$

The aggregate miss rate is

$$M_{victim} = 10\%$$

The hit rate of the main cache is

$$H_{main} = 0\%$$

The hit-rate of the victim is

$$H_{victim} = 90\%$$

Percentage of references that result in swaps between the main cache and the cache assist is

$$Swaps_{victim} = 90\%$$

Now let us consider the behavior of the proposed annex cache to this sequence. In the case of annex, A gets in the main cache and stays there. The reference B will be moved from the annex cache to the main cache only after two successive references without any intermediate reference to A. Since there are no two successive references to B, B stays in the annex cache all the time and A stays in the main cache eliminating the swaps between the two caches. Let us use A_{h-a} to represent a hit in the annex cache and A_{h-m} to represent a hit in the main cache. The behavior of the annex cache is

$$A_m B_m (A_{h-m} B_{h-a})^9$$

The aggregate miss rate is

$$M_{victim} = 10\%$$

The hit rate of the main cache is

$$H_{main} = 45\%$$

The hit-rate of the victim is

$$H_{annex} = 45\%$$

Percentage of references that result in swaps between the main cache and the cache assist is

$$Swaps_{annex} = 0\%$$

Thus it is seen that both annex and victim caches are significantly better than the conventional cache. Although the aggregate miss ratios for annex and victim are the same, victim caching involves swapping for 90% of the references whereas annex caching reduces the swapping to none.

2.3.2 Conflict between inner and outer loops

Now let us consider another case where there is a conflict between a reference inside a loop with another reference outside the inner loop. If the outer loop is executed 10 times and the inner loop is executed 10 times in each outer loop, the memory access pattern may be represented as $(A^{10}B)^{10}$. The behavior of a conventional cache would be

$$(A_m A_h^9 B_m)^{10}$$

and the miss rate of a conventional cache is

$$M_{conv} = 18\%.$$

Now let us consider the behavior of a victim cache to this sequence. In the conventional cache, A would be replaced by B and the first reference to A in every inner loop and the reference to B in the outer loop would be a miss. In the victim cache, the when A gets replaced by B, A would be put into the victim cache; hence after the initial miss to A and B, there are no further misses. The main cache misses will be filled from the victim cache. The behavior of a victim cache may be represented as

$$A_m A_{h-m}^9 B_m (A_{h-v} A_{h-m}^9 B_{h-v})^9$$

The aggregate miss rate is

$$M_{victim} = 2/110 = 1.8\%$$

The hit rate of the main cache is

$$H_{main} = 81.8\%$$

The hit-rate of the victim is

$$H_{victim} = 16.4\%$$

Percentage of references that result in swaps between the main cache and the cache assist is

$$Swaps_{victim} = 16.4\%$$

Now let us consider the behavior of the proposed annex cache to this sequence. In the case of annex, A gets in the main cache and stays there. The reference B will be moved from the annex cache to the main cache only after two successive references without any intermediate reference to A. Since there are no two successive references to B, B stays in the annex cache all the time and A stays in the main cache eliminating the swaps between the two caches. The behavior of the annex cache may be represented as

$$A_m A_{h-m}^9 B_m (A_{h-m}^{10} B_{h-a})^9$$

The aggregate miss rate is

$$M_{annex} = 2/110 = 1.8\%$$

The hit rate of the main cache is

$$H_{main} = 90\%$$

The hit-rate of the annex is

$$H_{annex} = 8.2\%$$

Percentage of references that result in swaps between the main cache and the cache assist is

$$Swaps_{annex} = 0\%$$

Thus one may observe that there are more main cache hits although the aggregate miss ratio stays the same. It may also be observed that there are no swaps in annex whereas 16.4% of the accesses result in swaps in the case of victim caching.

2.3.3 Conflict between loops

Now let us consider another case where there is a conflict between a reference inside two different loops. If there is such a conflict and there is a memory access pattern such as $(A^{10}B^{10})^{10}$. The behavior of a conventional cache would be

$$(A_m A_h^9 B_m B_h^9)^{10}$$

and the miss rate of a conventional cache is

$$M_{conv} = 10\%.$$

Now let us consider the behavior of a victim cache to this sequence. In the conventional cache, A would be replaced by B and the first reference to A in every loop and the first reference to B in every loop would be a miss. In the victim cache, the when A gets replaced by B, A would be put into the victim cache; hence after the initial miss to A and B, there are no further misses. The main cache misses will be filled from the victim cache. The behavior of a victim cache is represented as

$$A_m A_{h-m}^9 B_m B_{h-m}^9 (A_{h-v} A_{h-m}^9 B_{h-v} B_{h-m}^9)^9$$

The aggregate miss rate is

$$M_{victim} = 2/200 = 1\%$$

The hit rate of the main cache is

$$H_{main} = 90\%$$

The hit-rate of the victim is

$$H_{victim} = 9\%$$

Percentage of references that result in swaps between the main cache and the cache assist is

$$Swaps_{victim} = 9\%$$

Now let us consider the behavior of the proposed annex cache to this sequence. Reference A enters the main cache. For the first referencing of B it enters the annex cache but does not enter the main cache until the second reference. Hence the annex cache algorithm slightly deteriorates the performance by the extra inertia in letting B enter the main cache. The behavior of the annex cache may be represented as

$$A_m A_{h-m}^9 B_m B_{h-a} B_{h-m} (A_{h-a}^2 A_{h-m}^8 B_{h-a}^2 B_{h-m}^8)^9$$

The aggregate miss rate is

$$M_{annex} = 2/200 = 1\%$$

The hit rate of the main cache is

$$H_{main} = 80\%$$

The hit-rate of the annex is

$$H_{annex} = 18\%$$

Percentage of references that result in swaps between the main cache and the cache assist is

$$Swaps_{annex} = 9\%$$

In this example, the annex cache algorithm does not reduce the swaps between the main cache and the assist cache; it even decreases the main cache hit ratio by the inertia in letting entries go to the main cache.

3 Performance Evaluation

In this section, we evaluate the performance of the annex caching scheme using extensive trace-driven simulations on several large real benchmark programs. Trace-driven simulation has the advantage of fully and accurately reflecting real instruction streams. We also compare the performance to victim caching [11] and half and half caching [26].

3.1 Performance Metrics

The performance measure best suited for the evaluation of the proposed cache assist is effective memory access time or total memory access time. If h_m is the fraction of accesses which hit in the primary cache and h_a is the fraction of hits in the assist cache and h_{sw} is the number of hits in assist cache accompanied by swaps, if t_m and t_a are the access times of the main cache and assist cache, t_{sw} is the penalty for swapping, and t_{eff} is the effective access time, then

$$t_{eff} = h_m * t_m + h_a * t_a + h_{sw} * t_{sw} + (1 - h_m - h_a - h_{sw}) * misspenalty$$

We assume the main cache access time t_m to be 1 cycle, time for direct access of the assist cache t_a to be 2 cycles and the additional penalty for swapping t_{sw} to be 1 cycle. Thus t_a and t_m values are the same as that assumed by Theobald et al [26]. We have assumed a very small value for the swapping penalty. If the actual swap penalty is higher (depending on the swap hardware, datapath width and block size), an architecture that reduces the number of swaps will improve the performance even beyond the projections in this paper.

To facilitate easy comparison, we compute a speedup figure as the ratio of the effective access times without and with the cache assist.

$$Speedup = \frac{t_{eff} - with - no - cache - assist}{t_{eff} - with - cache - assist}$$

Benchmark	Total references	Distinct references
xlisp	18,408,337	19,337
ear	5,767,488	69,773
espresso	6,864,863	18,911
tomcatv	11,953,113	15,607
eqntott	467,730	42,253
ora	5,921,107	3,028
doduc	13,221,528	7,657
alvinn	6,689,213	54,780
compress	2,843,636	116,425
swm	7,093,038	858,676

Table 1: Characteristics of the data traces used

3.2 Benchmarks and Trace generation

The benchmarks consisted of several programs from the SPEC92 suite, *compress, xlisp, eqntott, espresso, doduc, ora, swm256, ear, alvinn,* and *tomcatv*. The address traces were generated by *pixie* [20] [24] on a DEC5000 workstation which uses the MIPS R3000 processor. The SPEC92 programs were compiled with default Make files and the pixified executables of the programs were generated and traced. In order to simulate the entire traces of the regular input files, several months of simulations will be required. Hence we decided to perform the simulations for only 30 million references or less. If SPEC provided a short input with less than 30 million references, we used that and traced the program entirely. The smaller inputs were used because then we could trace the entire application and see effects due to operating system activity as well. If such a small input was not available, the first 30 million references were traced. We have observed several researchers including [19] [9] deleting the initial several million references. We did the same for the data traces for which no small input files were available. Since these are not entire runs, the effect of the operating system activity would be disproportionate when we trace only a part of the complete trace.

Tables 1 and 2 illustrate the number of references in the data and instruction traces used for the simulation. We also indicate the number of distinct references so that one can know the size of the cache that will hold the entire program or data set.

3.3 Configurations simulated

We used cache sizes of 4k, 32k and 64k bytes, with a block size of 32 bytes. A victim cache or annex cache of 128 bytes is added to the aforementioned main cache size. Half and half caches as described in Theobald et al's paper [26] use the conventional victim cache algorithm. We performed experiments with such half and half caches. We also performed experiments with half and half caches where the secondary half of the cache is treated as an annex cache rather than a victim cache. Since the annex/victim cache size initially chosen was only 128 bytes, with our block size of 32 bytes, it is only 4 blocks. For data, we performed experiments with victim and annex caches of size 16 blocks also. A miss penalty of 40 cycles is assumed considering the block size of 32 bytes, bus width of 64 bytes, and access time of 10 processor cycles.

Benchmark	Input	Total references	Distinct references
ear	args.short	29,984,540	5,487
espresso	Z5xp1.in	28,918,812	16,575
tomcatv		21,305,640	5,051
eqntott	ex5.eqn	1,181,058	2,414
ora	params	18,806,280	6,077
doduc	doducin	29,775,327	23,497
swm256	swm256.in	29,442,094	6,114
alvinn		29,038,411	1,329
xlisp	li-input.lsp	30,000,001	5,073
compress	in (short)	8,636,179	1,748

Table 2: Characteristics of the instruction traces used

3.4 Performance of synthetic sequences

Before performing experiments with SPEC92 programs, we performed some experiments with a few synthetic trace sequences similar to the examples described in the previous section. Table 3 illustrates a few reference patterns and details about the number of hits in main and annex and victim caches. In Table 4, the aggregate miss ratios for these small sequences are presented. Although the aggregate miss rates are same for victim and annex caches, the performance is different due to the difference in the number of swaps. This is clear from the total access times presented in Table 5 for a miss penalty of 40 cycles. From Table 4 and Table 5, it is seen that except patterns 2, 9 and 11, all patterns yield improved performance with the proposed annex caching policy. It may also be noted that both annex and victim perform significantly better than conventional caches.

The speedups for the benchmark programs are presented in Fig. 3.

3.5 Performance of SPEC92 programs

The performance of annex caches for real programs will depend on the actual memory referencing patterns in the particular program. Fig. 4 illustrates the aggregate miss rates in conventional caches, annex caches of size 4 blocks, victim caches of size 4 blocks, half and half caches with annex caching policy, half and half caches with victim caching policy, annex caches of size 16 blocks and victim caches of size 16 blocks. In *swm*, 4-block annex case yields extremely poor performance even compared to conventional cache. This was due to heavy conflicts within annex and the problem is not present when the annex size is 16 blocks. For size of 16 blocks, annex performs equal or better than the victim cache. In many cases the miss ratios fall to zero and are not visible in the graphs. This happens more frequently in the larger caches.

Fig 5 illustrates the speedup of annex caches of size 4 blocks, victim caches of size 4 blocks, half and half caches with annex caching policy, half and half caches with victim caching policy, annex caches of size 16 blocks and victim caches of size 16 blocks, compared to conventional cache. Annex cache of size 4 blocks yields very poor performance however annex caches of size 16 blocks or half and half caches often perform equal or better than victim caches. However, annex caches do not display any dramatic improvement as in the synthetic sequences.

	Reference pattern	H_{conv}	$HMain_a$	$HMain_v$	$Hits_a$	$Hits_v$	Sw_a	Sw_v
1	$(A^{10}B)^{10}$	90	99	90	9	18	0	18
2	$(A^{10}B^{10})^{10}$	180	161	180	37	18	19	18
3	$(ABB)^{10}$	10	18	10	10	18	1	18
4	$(BAB)^{10}$	9	19	9	9	19	0	19
5	$(AAAB)^{10}$	20	29	20	9	18	0	18
6	$(ABC)^{10}$	0	1	0	26	27	13	27
7	$(ABB^{10})^{10}$	100	108	100	10	18	1	18
8	$(BAB^{10})^{10}$	99	109	99	9	19	0	19
9	$(AAAB^{10})^{10}$	111	92	111	37	18	19	18
10	$(ABC^{10})^{10}$	91	89	91	29	27	11	27
11	$(A^{10}B^{10}C^{10}D^{10}E^{10})^{10}$	450	401	450	49	0	49	0
12	$(ABACADAEAF)^{10}$	0	49	0	0	49	0	49
13	$(AB)^{10}$	0	9	0	9	18	0	18

Table 3: Number of hits, misses and swaps in main cache and annex/victim caches for a few reference patterns

	Reference Pattern	M_{conv}	M_{annex}	M_{victim}
1	$(A^{10}B)^{10}$	18	1.8	1.8
2	$(A^{10}B^{10})^{10}$	10	1.0	1.0
3	$(ABB)^{10}$	67	6.7	6.7
4	$(BAB)^{10}$	70	6.7	6.7
5	$(AAAB)^{10}$	50	5	5
6	$(ABC)^{10}$	100	10	10
7	$(ABB^{10})^{10}$	16.7	1.7	1.7
8	$(BAB^{10})^{10}$	17.5	1.7	1.7
9	$(AAAB^{10})^{10}$	15.2	1.7	1.7
10	$(ABC^{10})^{10}$	25	2.5	2.5
11	$(A^{10}B^{10}C^{10}D^{10}E^{10})^{10}$	10	10	10
12	$(ABACADAEAF)^{10}$	100	51	51
13	$(AB)^{10}$	100	10	10

Table 4: Aggregate miss ratios of conventional cache, annex cache and victim cache for a few reference patterns

	Reference Pattern	M_{conv}	M_{annex}	M_{victim}
1	$(A^{10}B)^{10}$	890	197	224
2	$(A^{10}B^{10})^{10}$	980	334	314
3	$(ABB)^{10}$	810	119	144
4	$(BAB)^{10}$	849	117	146
5	$(AAAB)^{10}$	820	127	154
6	$(ABC)^{10}$	1200	186	201
7	$(ABB^{10})^{10}$	900	209	234
8	$(BAB^{10})^{10}$	939	207	236
9	$(AAAB^{10})^{10}$	911	265	245
10	$(ABC^{10})^{10}$	1291	278	292
11	$(A^{10}B^{10}C^{10}D^{10}E^{10})^{10}$	2450	2548	2450
12	$(ABACADAEAF)^{10}$	4000	2089	2187
13	$(AB)^{10}$	800	107	134

Table 5: Total access times for the conventional cache, annex cache and victim cache for a few reference patterns. A miss penalty of 40 cycles was assumed.

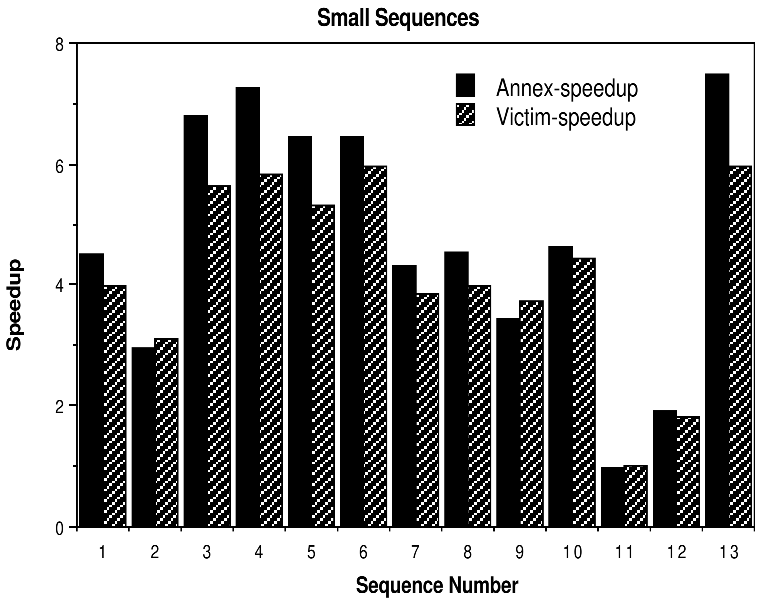

Figure 3: Speedup of annex and victim caches compared to conventional caches

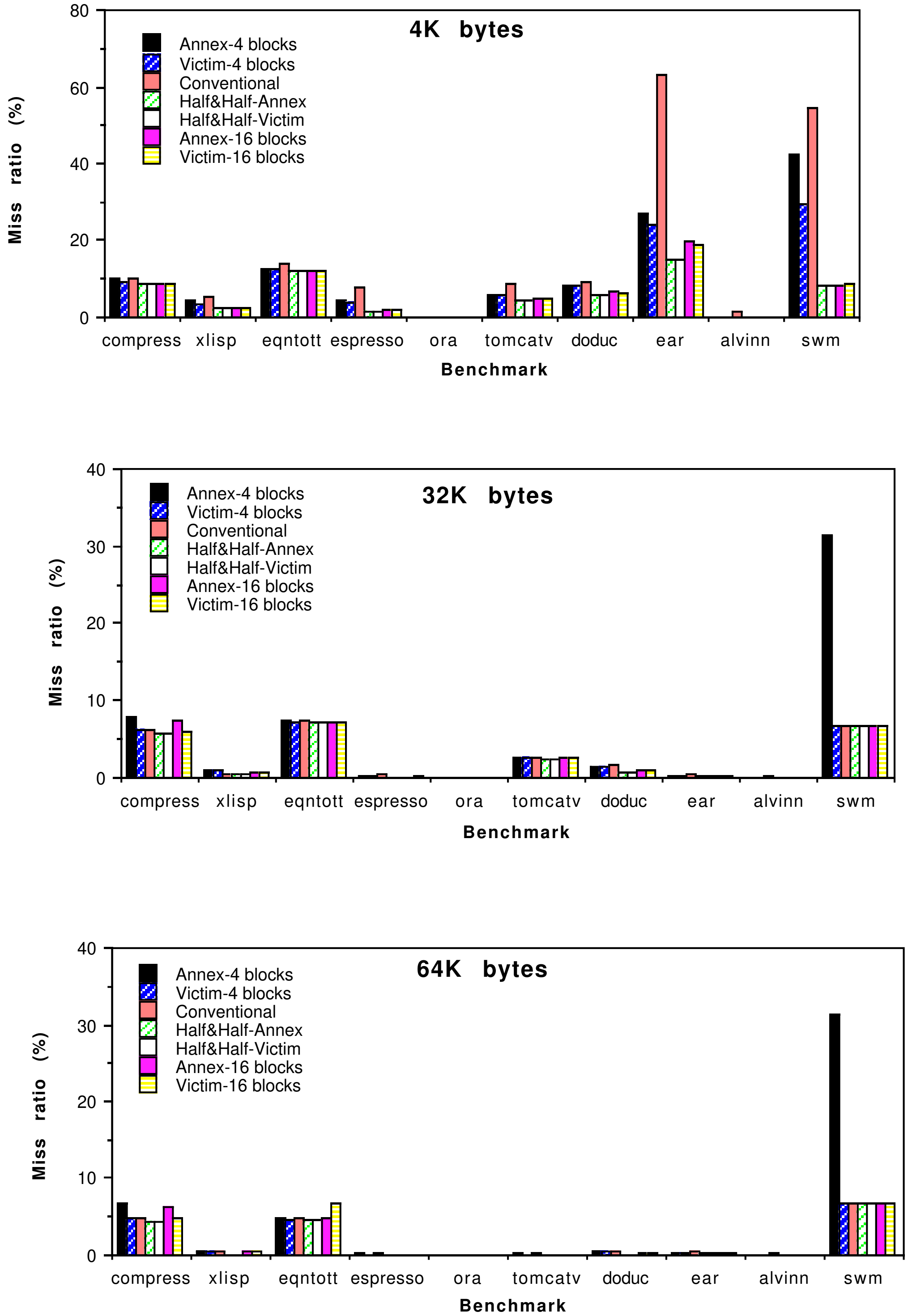

Figure 4: Aggregate miss rates in conventional, annex and victim caches for data traces

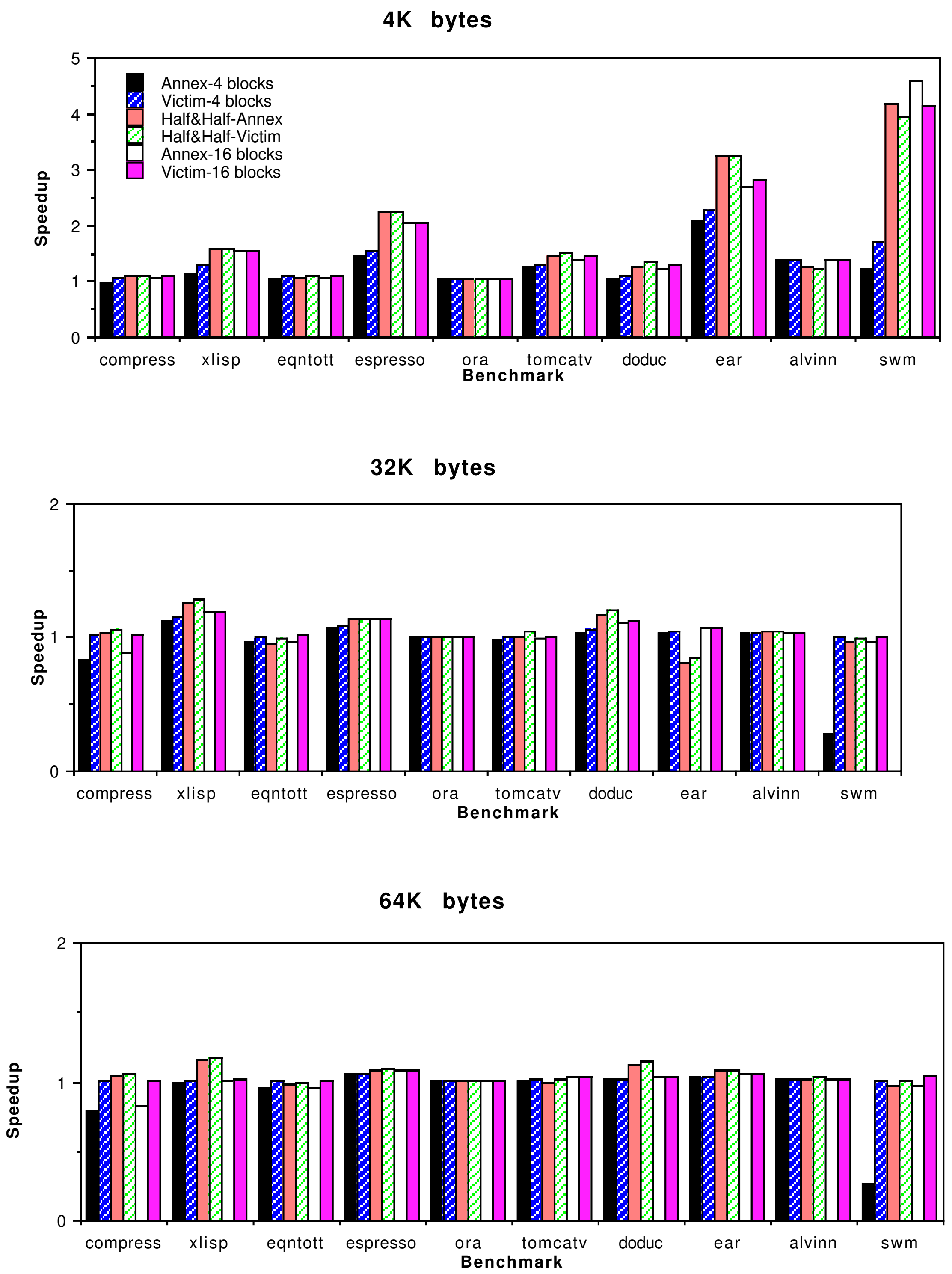

Figure 5: Speedup in data access time for annex and victim caches compared to conventional cache.

	Configurations	4k bytes	32k bytes	64k bytes
Data	Annex	1.810	1.035	0.999
	Victim	1.797	1.063	1.032
	Half and half - annex	1.843	1.036	1.046
	Half and half - victim	1.842	1.062	1.057
Instr	Annex	1.053	1.055	0.999
	Victim	1.037	1.026	1.0
	Half and half - annex	1.322	1.220	1.022
	Half and half - victim	1.355	1.232	1.027

Table 6: Aggregate speedup over all benchmarks

Fig. 6 illustrate the miss ratios and Fig 7 illustrate the speedups for instruction traces. The annex/victim cache size is 4 blocks. Since we are comparing the annex and victim caches to conventional caches, the comparison is more fair when the assist cache size is smaller.

The aggregate speedups from all the benchmarks are summarized in Table 6 for instruction and data traces. Half and half caching is a clear winner in the case of instructions and winner with no significant advantage in the case of data. There is no clear winner between the annex and victim algorithms either in the small annex/victim or the half and half case. However, it may be remembered that annex caches do not need as elaborate swap hardware as victim caches.

4 Summary and Conclusion

In this paper, we presented the design of a cache assist namely annex cache that alleviates conflict misses and implements selective caching. The performance of the proposed annex cache was evaluated using trace-driven simulation, for several programs in the SPEC92 benchmark suite. The performance of the annex cache is almost always better than the equivalent conventional cache. We also compared the performance of the annex cache to similar cache assists such as the victim cache and half and half cache. The annex cache yields a performance comparable to victim caches, but using less sophisticated swap hardware. We are in the process of tuning the annex caching algorithm to further improve its performance.

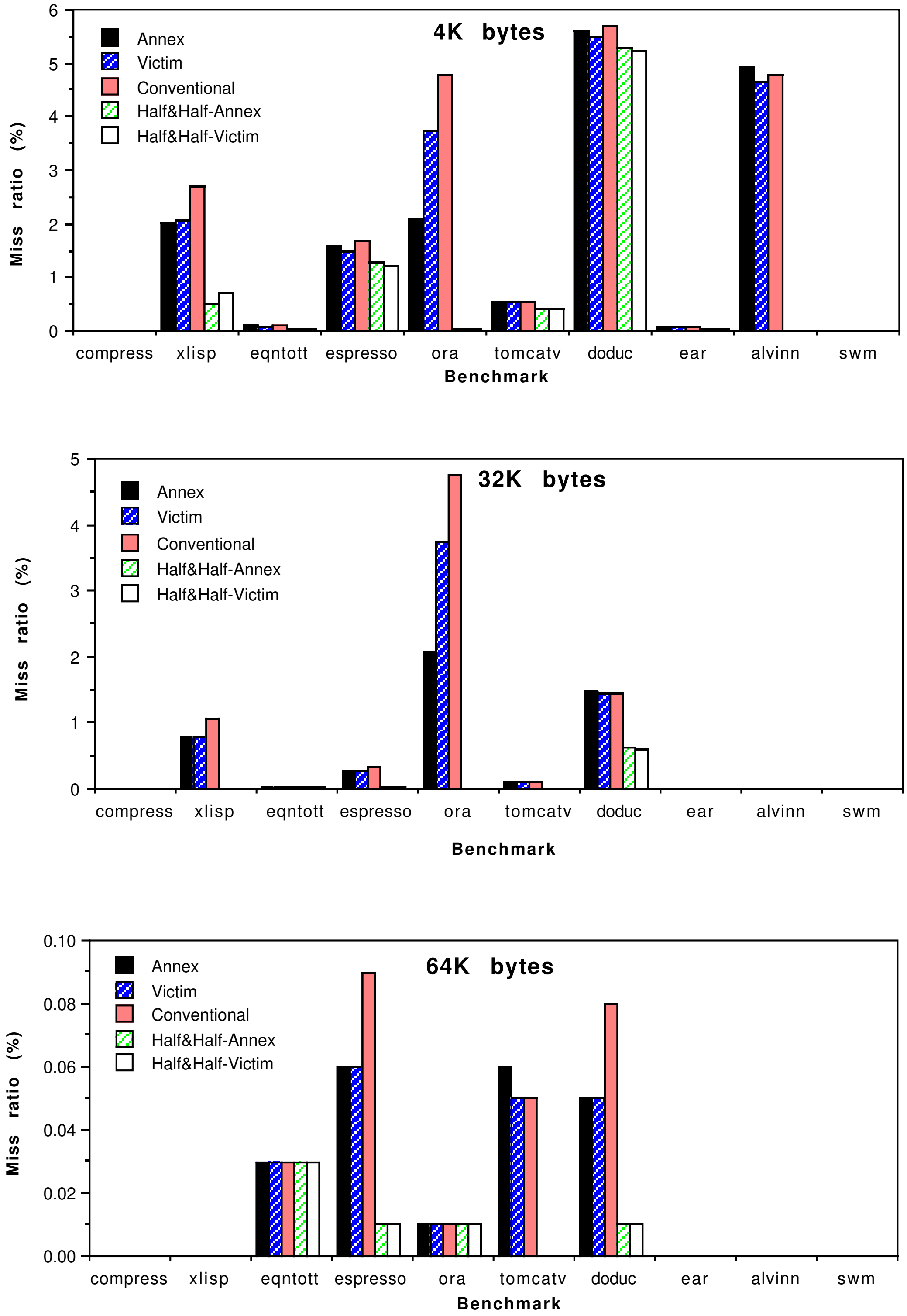

Figure 6: Aggregate miss rates in conventional, annex and victim caches for instruction traces

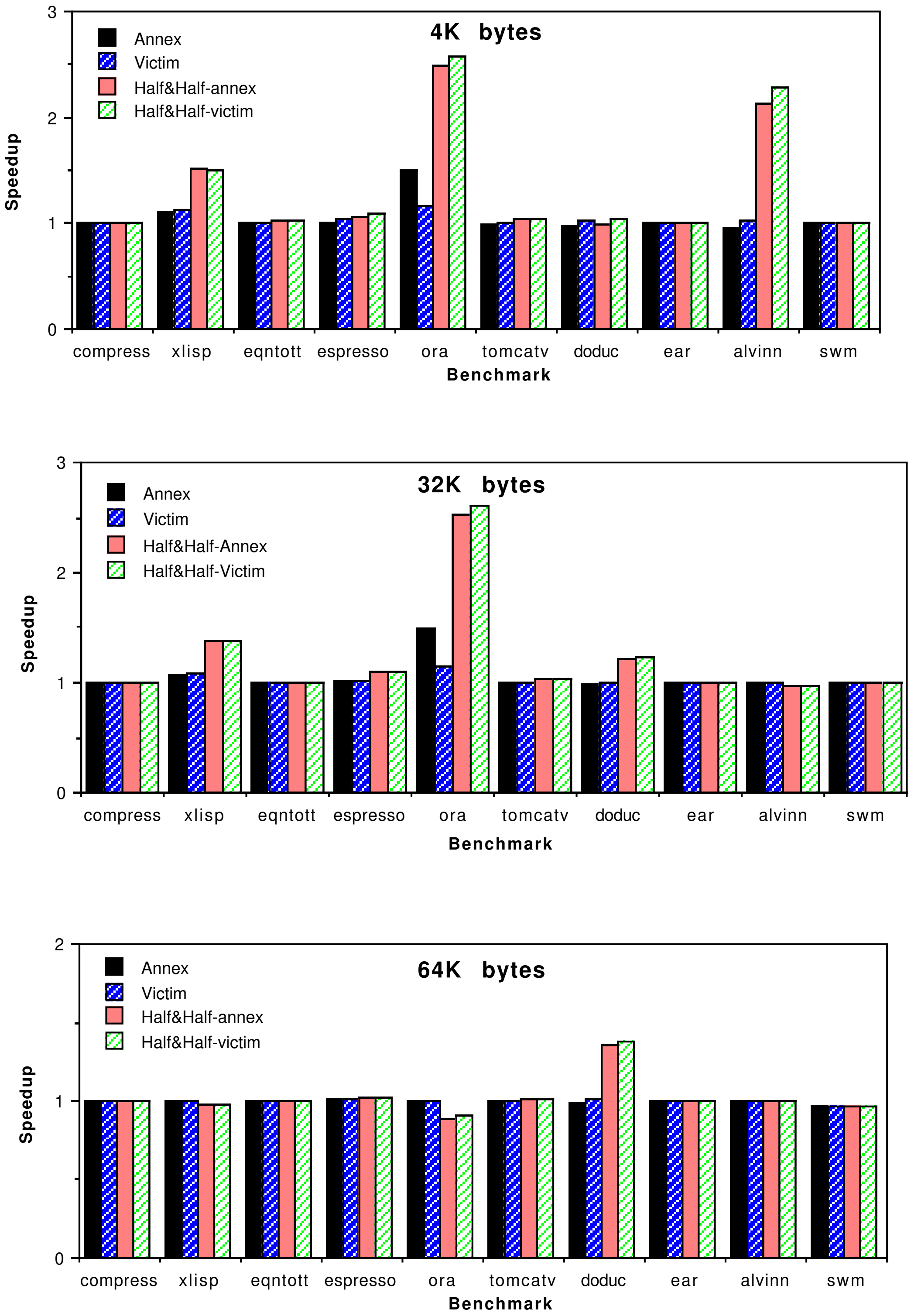

Figure 7: Speedup in instruction access time for annex and victim caches compared to conventional caches

References

[1] S. G. Abraham, R. S. Sugumar, B. R. Rau and Rajiv Gupta, "Predictability of load/store instruction latencies", In Proceedings of MICRO-26, 1993, pp. 139-152.

[2] A. Agarwal, J. Hennessy, and M. Horowitz, "Cache Performance of Operating Systems and Multiprogramming", ACM Transactions on Computer Systems, 6(4):393-431, November 1988.

[3] A. Agarwal and S. Pudar, "Column-Associative Caches: A Technique for Reducing the Miss Rate of Direct-Mapped Caches", In International Symposium on Computer Architecture, 1993, pp. 179-190.

[4] J-L Baer and W-H Wang, "On the Inclusion Properties for Multi-Level Cache Hierarchies", The 15th Annual Symposium on Computer Architecture, IEEE Computer Society Press, June 1988, pp. 73-80.

[5] B. Bershad, D. Lee, T. Romer, B. Chen, "Avoiding Conflict Misses Dynamically in Large Direct-Mapped Caches", Proceedings of ASPLOS-VI, pp. 158-170.

[6] J. H. Chang, H. Chao, and K. So, "Cache Design of a Sub-Micron CMOS System/370", in Proceedings of the 14th Annual International Symposium on Computer Architecture, Pittsburg, Pennsylvania, pp. 208-213, June 1987.

[7] W. Y. Chen, P. P. Chang, T. M. Conte, and W. W. Hwu, "The Effect of Code Expanding Optimizations on Instruction Cache Design", IEEE Transactions on Computers, 42(9):1045-1057, September 1993.

[8] C-H Chi ad H. Dietz, "Unified Management of Registers and Cache Using Liveness and Cache Bypass", Proceedings of the ACM SIGPLAN '89 Conference on Programming Language Design and Implementation, Vol. 24(7), pp. 344-355, June 1989.

[9] A. Gonzalez, C. Aliagas, and M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality", Proceedings of the ACM 1995 International Conference on Supercomputing, Spain, pp. 338-347.

[10] N. P. Jouppi, "Architectural and Organizational Tradeoffs in the Design of the MultiTitan CPU", The 16th Annual Symposium on Computer Architecture, IEEE Computer Society Press, May 1989, pp. 281-289.

[11] N. P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Buffers", Proceedings of the 17th International Symposium on Computer Architecture, 1990, pp. 364-373.

[12] N. P. Jouppi and S. J. E. Wilton, "Trade-offs in Two-Level On-Chip Caching", Proceedings of the 21st International Symposium on Computer Architecture, Chicago, Illinois, pp. 34-45, April 1994.

[13] D. R. Kerns and S. J. Eggers, "Balanced Scheduling: Instruction Scheduling when memory latency is uncertain", Proceedings of the SIGPLAN 93 Conference on Programming Language Design and Implementation, pp. 278-289.

[14] R. E. Kessler, M. D. Hill, and D. A. Wood, “A Comparison of Trace-Sampling Techniques for Multi-Megabyte Caches”, IEEE Transactions on Computers, Vol. 43, No. 6, June 1994, pp. 664-675.

[15] M. S. Lam, E. E. Rothberg and M. E. Wolf, “The Cache Performance and Optimizations of Blocked Algorithms”, Proceedings of the Fourth International Conference on Architectural Support for Programming Languages and Operating Systems”, 1991, pp. 63 – 74.

[16] A. R. Lebeck and D. A. Wood, “Cache Profiling and the SPEC Benchmarks: A Case Study”, IEEE Computer, October 1994, pp. 15-26.

[17] A. Mendlson, S. Pinter, and R. Shtokhamer, “Compile Time Instruction Cache Optimizations”, In Computer Architecture News, pp. 44-51, March 1994.

[18] S. McFarling, “Program Optimization for Instruction Caches”, Proceedings of the Third International Conference on Architectural Support for Programming Languages and Operating Systems”, 1989, pp. 183 – 191.

[19] S. McFarling, “Cache Replacement with Dynamic Exclusion”, Proceedings of International Symposium on Computer Architecture, May 1992, pp. 191-200.

[20] UMIPS-V Reference Manual, MIPS Computer Systems, Sunnyvale, California, 1990.

[21] S. Palacharla and R.E. Kessler, “Evaluating Stream Buffers as a Secondary Cache replacement”, Proceedings of the 21st International Symposium on Computer Architecture, April 1994, pp. 24-33.

[22] A. Seznec, “ A Case for Two-way skewed Associative Caches”,Proceedings of the 20th International Symposium on Computer Architecture, pp. 169-178, 1993.

[23] Steven Przybylski, M. Horowitz, and J. Hennessy, “Performance Tradeoffs in Cache Design”, Proceedings of the 15th Annual Symposium on Computer Architecture, pp. 290-298, IEEE Computer Society Press, June 1988.

[24] Michael D. Smith, “Tracing with pixie”, Technical Report No. CSL-TR-91-497, Computer Systems Laboratory, Stanford University, Stanford, California.

[25] O. Temam and N. Drach, “Software Assistance for Data Caches”, Proceedings of the High Performance Computer Architecture Symposium, Jan 1995, pp. 154-163.

[26] K. B. Theobald, H. H. J. Hum, and G. R. Gao, “A Design Framework for Hybrid-Access Caches”, Proceedings of the First High Performance Computer Architecture Symposium, Raleigh, January 1995, pp. 144-153.

[27] J. Torrellas, C. Xia, and R. Daigle, “Optimizing Instruction Cache Performance for Operating System Intensive Workloads”, Proceedings of the High Performance Computer Architecture Symposium, Jan 1995, pp. 360-369.

[28] K. Westerholz, Stephan Honal, J. Plankl, and C. Hafer, “Improving Performance by Cache Driven Memory Management”, Proceedings of the International High Performance Computer Architecture Symposium, Jan 1995, pp. 234-242.