Microprocessors and Microsystems 23 (1999) 537–551

MICROPROCESSORS AND MICROSYSTEMS

www.elsevier.nl/locate/micpro

Annex cache: a cache assist to implement selective caching[☆]

L.K. John[a,*], T. Li[a], A. Subramanian[b]

[a]Electrical and Computer Engineering Department, University of Texas at Austin, Austin, TX 78712, USA
[b]Computer Science and Engineering Department, University of South Florida, Tampa, FL 33620, USA

Received 1 September 1998; received in revised form 8 July 1999; accepted 8 July 1999

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 (N.P. Jouppi, Improving direct-mapped cache performance by the addition of a small fully associative cache and buffers, Proceedings of the International Symposium on Computer Architecture, 1990, pp. 364–373) but the processor can access annex cache entries directly, i.e. annex cache entries can bypass the main cache. Thus it combines the features of victim caches and cache exclusion schemes. Extensive simulation studies for annex and victim caches using a variety of SPEC programs are presented in this paper. Annex caches were observed to be significantly better than conventional caches, better than victim caches in certain cases, and comparable to victim caches in other cases.
© 1999 Elsevier Science B.V. All rights reserved.

Keywords: Cache misses; Conflict misses; Cache exclusion; Cache bypassing; Cache replacement; Victim caches

1. Introduction

While designing cache systems, there are several trade-offs involving mapping schemes, replacement strategies, fetch policies etc. The average access time for a memory reference in a system with cache depends on the hit-rate (or the miss-rate) and the miss penalty. Hence, improving cache performance involves minimizing the miss-rate and the miss penalty. Increasing the block size often reduces the miss-rate, however this results in increasing the miss penalty. The miss-rate can also be reduced by increasing the cache size, and/or the associativity of the cache. However, it is difficult to significantly increase the cache size or associativity while matching the cache hit access time to the clock speeds of modern fast processors. Many current microprocessors run at clock speeds ranging from 500 to 700 MHz, and Gigahertz microprocessors are on the way. It is extremely difficult to design large associative caches with hit-access times that match such clock speeds. Small direct-mapped caches are preferred under these conditions. Direct-mapped caches perform better than set-associative caches when the access time costs for hits are considered [1]. 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. This paper deals with improving the performance of direct-mapped instruction caches using cache bypassing or 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

[☆]Based on ''Design and Performance Evaluation of a Cache Assist to Implement Selective Caching'' by L. John and A. Subramanian, which appeared in Proceedings of the International Conference on Computer Design, 1997, pp. 510–518, ©1997 IEEE.

*Corresponding author. Tel: + 1-512-232-1455.
E-mail address: ljohn@ece.utexas.edu (L.K. John)

miss problem can be alleviated by excluding infrequent instructions from the cache. The annex cache implements bypassing or cache exclusion for infrequent items. 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 here is that an item deserves to be in the cache only if it can prove its right to exist in the cache.

1.1. Related research

To deal with the high miss-rates in direct mapped caches, designers and architects have developed various schemes that combine fast access with greater associativity by either providing pseudo-associativity or by having a fast direct-mapped primary section together with a secondary section of higher associativity. Some examples of schemes to improve performance of direct mapped caches are victim caches [1], hash–rehash caches [2], column-associative caches [3], MRU caches [4], half-and-half caches [5], conflict-avoiding caches [6], and pollution control caches [7].

A victim cache [1], is a small fully associative cache of typically no more than 16 blocks. If a block in the main direct-mapped cache has to be replaced, that entry (called the *victim*) is transferred to the victim cache. The intent is that the next time there is a need for this entry, it can be accessed from the victim cache faster than if it would have been only in the next lower level of cache. During eviction of a victim from the main cache, if the desired block is in the victim cache, then that block swaps places with the victim during the second cycle.

The MRU cache [4], resulted from the observation that in a set-associative cache, the vast majority of hits within a set involve the most recently used (MRU) member of that set. In an MRU cache, during a read, all tags in the set are compared simultaneously, but the MRU block in that set is sent out immediately. If the tag for this item does not match the address tag, the data is invalidated. If any of the tags matches, then that block is latched into the output buffer making it the new MRU element of that set. Thus an MRU hit has an access time of one cycle, and a non-MRU hit takes two cycles.

In the aforementioned cache enhancements, some accesses take two cycles and some only one cycle. A simple way to combine fast access with better associativity is to read a direct-mapped cache twice. The hash-rehash cache [2], and the column-associative cache [3], exploit this principle. If a read misses on the first cycle, a second read is done using a different hashing function. If the second try is a hit, the first and second blocks are swapped. In the column-associative cache, a rehash bit is kept along with each tag, indicating whether that corresponding block represents a primary hit or a rehash hit. This bit is used to limit the swapping in order to reduce thrashing effects.

Half-and-half caches [5], are on-chip caches where half of the main cache area is used as an assist cache with the victim cache algorithm. Or in other words, one half of the cache is direct-mapped and the other half is associative. Theobald et al. found that in such a case where the assist cache is as big as the primary cache, the assist cache can simply be 2-way or 4-way set associative rather than fully-associative.

The conflict-avoiding cache [6], is based on polynomial modulus functions and demonstrates that pseudo-randomly indexed caches are effective in performance terms and practical from an implementation viewpoint. Pollution control caching from Walsh and Board [7] is another effort to improve performance of caches by controlling pollution of caches by non-conventional replacement algorithms.

Typically whenever there is a cache miss, the newly requested information enters the cache. Cache bypassing schemes typically passes the information directly from the lower layers in the memory hierarchy avoiding replacement of existing elements in the cache. McFarling [8] 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 directly 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 [10], Abraham et al. [11], and others. Chi and Dietz [10] showed that bypassing the cache can avoid cache pollution and improve performance for data loads and stores. Abraham et al. [11] 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.

Bershad et al. [12] 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 [13] 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 annex cache can directly pass the data to the processor bypassing the main cache, thereby eliminating the need for swapping in many cases. 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 directly passed 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. But if the swap data path is not as wide as the cache block size, several cycles will be needed to actually perform the swap.

McFarling's finite-state-machine [8] 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 [14]. 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 [8] 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 [8] or other research on selective caching [10,11]. We perform a quantitative comparison of direct-mapped, set-associative, victim and annex caches. Topham et al. [6] presented a polynomial mapping scheme that yields better performance than conventional mapping schemes, however no performance comparison with any of the pseudo-associative caches or assist caches was provided.

Major contributions of this paper include extensive simulation results that demonstrate the impact of bypassing in a cache assist. Although annex caches are clearly superior to conventional caches, the improvement over victim caches is not dramatic. Irrespective of this, the results in the paper are valuable because it demonstrates what kind of performance can be obtained by incorporating bypassing in a cache assist like the victim cache.

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 (Fig. 1). 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 [5] and Jouppi and Wilton's split-level caches [14].

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 on-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 [15] in cache hierarchies. As in victim caching, no item appears both in the main cache and the annex cache.

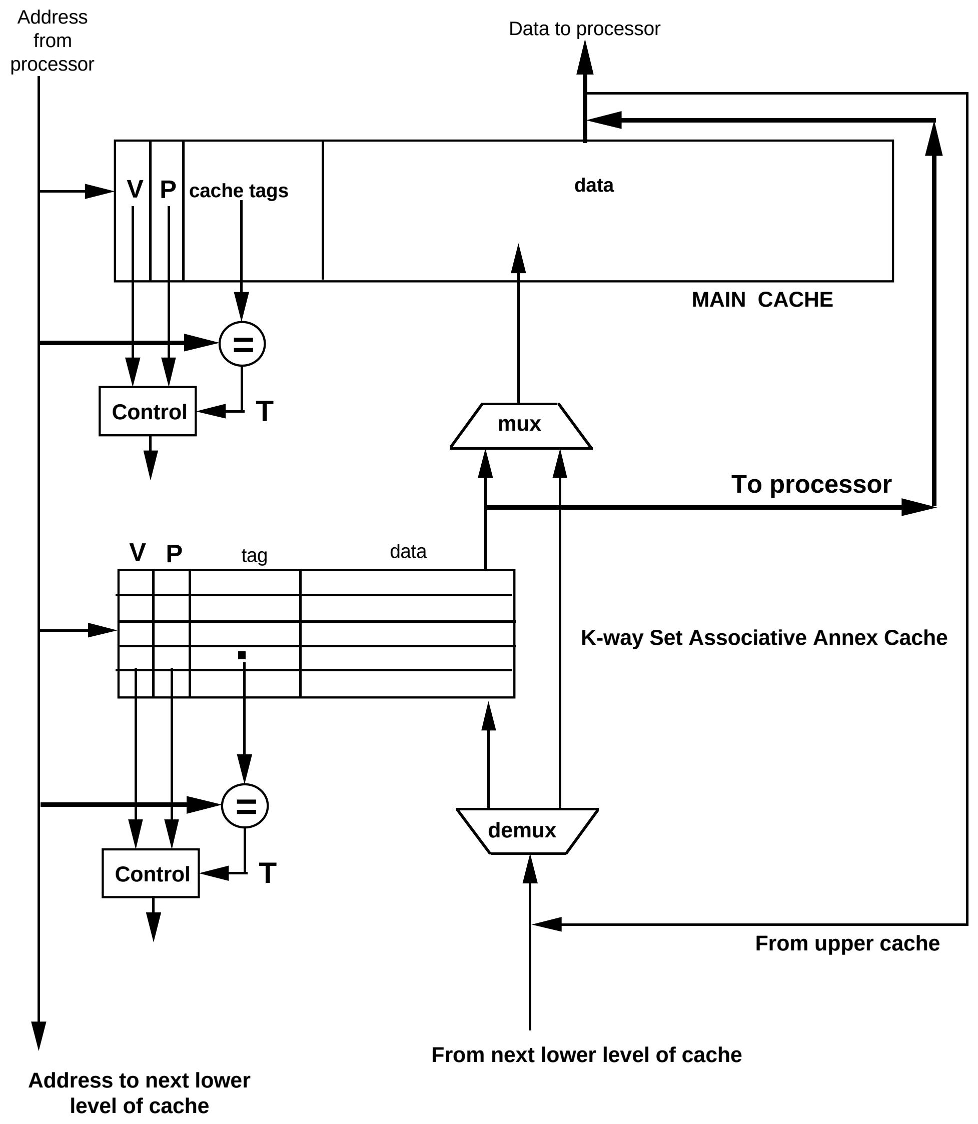

Fig. 1. Annex cache organization.

2.1. Algorithm

The annex cache operates as follows:

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 matches, 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 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

1. At cold start should items come through the annex or

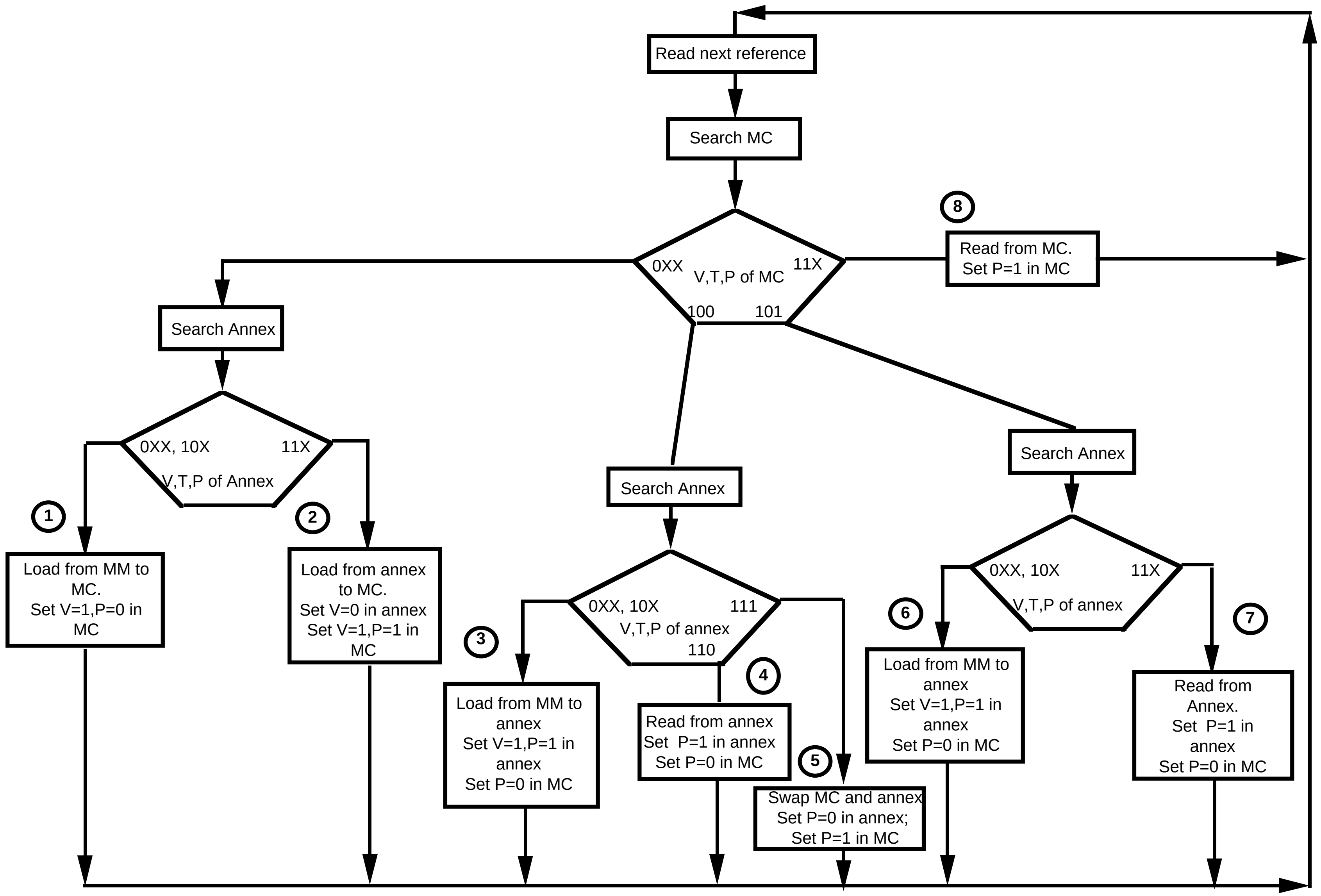
Read next reference
Search MC
V,T,P of MC
0XX
11X
100
101
8
Read from MC. Set P=1 in MC
Search Annex
V,T,P of Annex
0XX, 10X
11X
1
Load from MM to MC. Set V=1,P=0 in MC
2
Load from annex to MC. Set V=0 in annex Set V=1,P=1 in MC
Search Annex
V,T,P of annex
0XX, 10X
111
110
3
Load from MM to annex Set V=1,P=1 in annex Set P=0 in MC
4
Read from annex Set P=1 in annex Set P=0 in MC
5
Swap MC and annex Set P=0 in annex; Set P=1 in MC
Search Annex
V,T,P of annex
0XX, 10X
11X
6
Load from MM to annex Set V=1,P=1 in annex Set P=0 in MC
7
Read from Annex. Set P=1 in annex Set P=0 in MC

Fig. 2. Algorithm for the annex cache.

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.
2. 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.
3. 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 [8] 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, the two instructions 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 $A_{h\text{-}v}$ to represent that A is a hit in the victim cache and $A_{h\text{-}m}$ 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

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

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

$$\text{Swaps}_{victim} = 90\%$$

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 $A_{h\text{-}a}$ to represent a hit in the annex cache and $A_{h\text{-}m}$ to represent a hit in the main cache. The behavior of the annex cache is

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

The aggregate miss-rate is

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

The hit-rate of the main cache is

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

The hit-rate of the annex cache is

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

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

$$\text{Swaps}_{annex} = 0\%$$

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, 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

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

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

$\text{Swaps}_{victim} = 16.4\%$

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

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

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 cache is

$H_{annex} = 8.2\%$

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

$\text{Swaps}_{annex} = 0\%$

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, 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

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

Table 1
Basic characteristics of the traces—length of the instruction trace, length of the data trace, number of distinct instructions referenced and number of distinct data elements referenced are indicated

Benchmarks	Total I-refs.	Total D-refs.	Unique I-refs.	Unique D-refs.
alvinn	29038411	6689213	1329	54780
compress	8636179	2843636	1748	116425
doduc	29775327	13221528	23497	7657
ear	29984540	5767488	5487	69773
eqn	1181058	467730	2414	42253
espresso	28918812	6864863	16575	18911
ora	18806280	5921107	6077	3028
swm	29442094	7093038	6114	858676
tomcatv	21305640	11953113	5051	15607
xli	30000001	18408337	5073	19337
interleaved	221 million	73 million	70951	1164194

Table 2
Number of hits, misses and swaps in main cache and annex/victim caches for a few reference patterns

	Reference pattern	H_{conv}	$H\text{Main}_a$	$H\text{main}_v$	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

The aggregate miss-rate is

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

The hit-rate of the main cache is

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

The hit-rate of the victim is

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

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

$\text{Swaps}_{\text{victim}} = 9\%$

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

$A_m A^9_{h\text{-}m} B_m B_{h\text{-}a} B_{h\text{-}m} (A^2_{h\text{-}a} A^8_{h\text{-}m} B^2_{h\text{-}a} B^8_{h\text{-}m})^9$

The aggregate miss-rate is

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

The hit-rate of the main cache is

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

The hit-rate of the annex cache is

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

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

$\text{Swaps}_{\text{annex}} = 9\%$

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 [1] and half-and-half caching [5].

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 main 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, t_m and t_a are the access times of the main cache and assist cache, t_{sw} is the penalty for swapping,

Table 3
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

	Reference pattern	T_{conv}	T_{annex}	T_{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

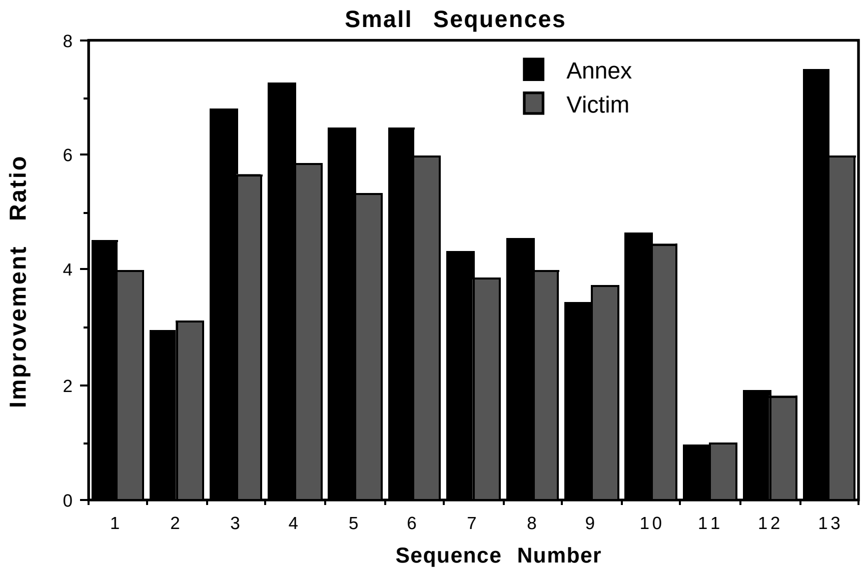

Fig. 3. Improvement in AMAT of annex and victim caches compared to conventional caches.

and t_{eff} is the effective access time, then

$$t_{\mathrm{eff}} = h_{\mathrm{m}} * t_{\mathrm{m}} + h_{\mathrm{a}} * t_{\mathrm{a}} + h_{\mathrm{sw}} * t_{\mathrm{sw}} + (1 - h_{\mathrm{m}} - h_{\mathrm{a}}) * \textit{miss penalty}$$

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.

To facilitate easy comparison, we compute the improvement in average memory access time (AMAT) as the ratio of the effective access times without and with the cache assist.

$$\text{Improvement ratio} = \frac{t_{\mathrm{eff}}\ (\text{with no cache assist})}{t_{\mathrm{eff}}\ (\text{with cache assist})}$$

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*. In addition, we create a trace interleaving various SPEC traces to simulate context switches that would occur in a multiprogrammed environment. This trace is marked *interleaved* in the graphs and tables. The address traces were generated by *pixie* [16,17] on a DEC5000 workstation which uses the MIPS R3000 processor. The SPEC programs were compiled with default Make files and the pixified executables of the programs were generated and traced. Table 1 illustrates the basic characteristics of the traces such as their length, number of unique instructions, number of unique data references, etc.

3.3. Configurations simulated

We used cache sizes of 4, 32 and 64 kbytes, with a block size of 32 bytes. A victim cache or annex cache of 16 bytes is added to the aforementioned main cache size. We also performed simulations with half-and-half caches as described in Theobald et al.'s paper [5]. Theobald et al.'s half-and-half caches are multilayer on-chip caches where the second layer is as fast as the first layer but there is no inclusion. The second layer is associative and behaves like a victim cache, except that in the original victim cache proposals, the victim cache is extremely small. 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. We also performed experiments with victim and annex caches of size 4 blocks also. A miss penalty of 40 cycles is assumed considering the block size of 32 bytes, and bus width of 64 bits.

3.4. Performance of synthetic sequences

Before performing experiments with SPEC programs, we performed some experiments with a few synthetic trace sequences similar to the examples described in the previous section. Table 2 illustrates a few reference patterns and details about the number of hits in main and annex and victim caches. 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 3 for a miss penalty of 40 cycles. From Table 3, it is seen that except patterns 2, 9 and 11, all patterns yield improved performance with the proposed annex caching policy. It may

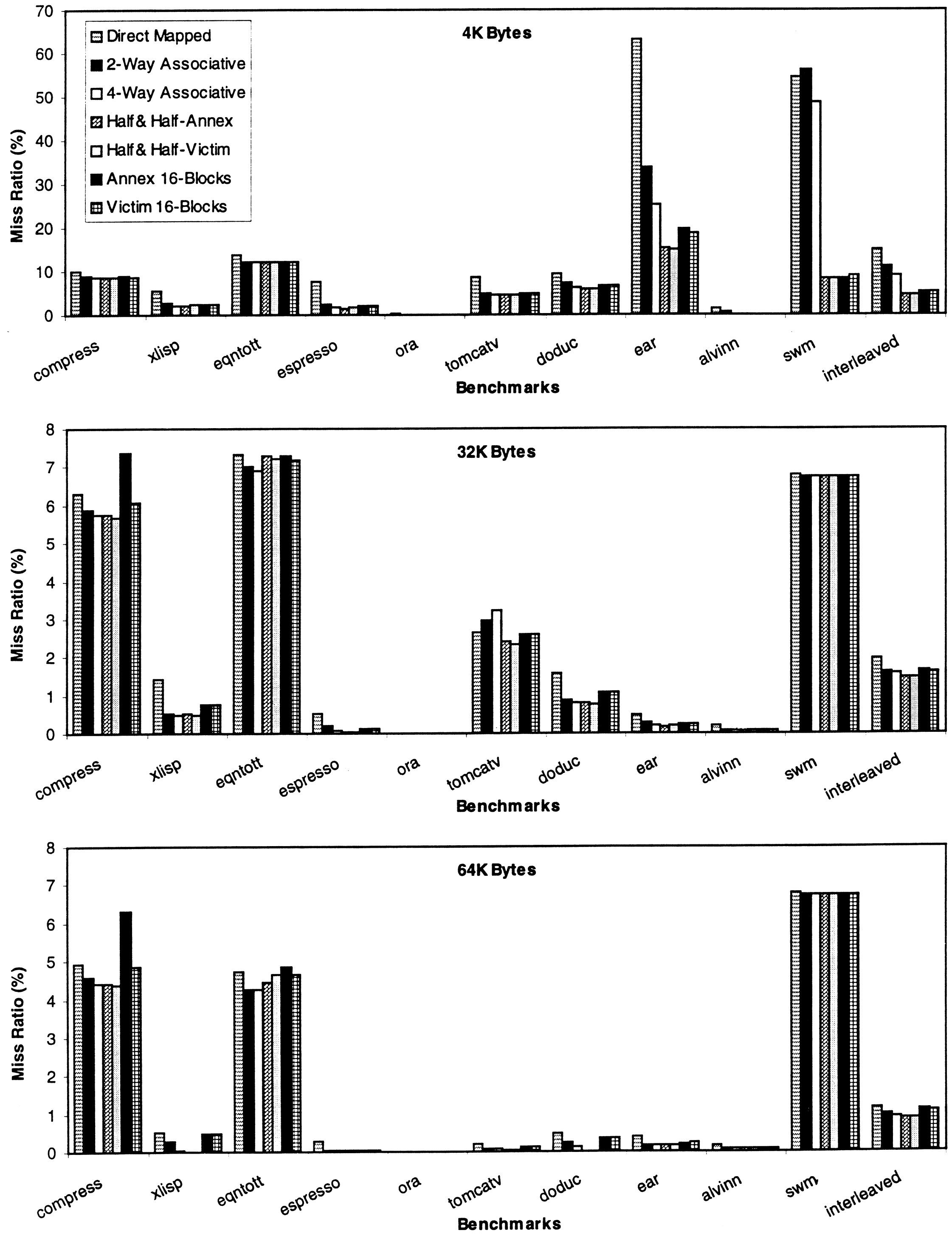

Fig. 4. Aggregate miss-rates in conventional, annex and victim caches for data traces.

also be noted that both annex and victim perform significantly better than conventional caches.

The improvement in access time for the benchmark programs are presented in Fig. 3.

3.5. Performance of SPEC 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 data miss-rates in conventional direct-mapped caches, 2- and 4-way set-associative caches, annex caches of size 16 blocks, victim caches of size 16 blocks, half-and-half caches with annex caching policy, half-and-half caches with victim caching policy. In most cases, the annex caches perform better than direct-mapped and even 2- or 4-way

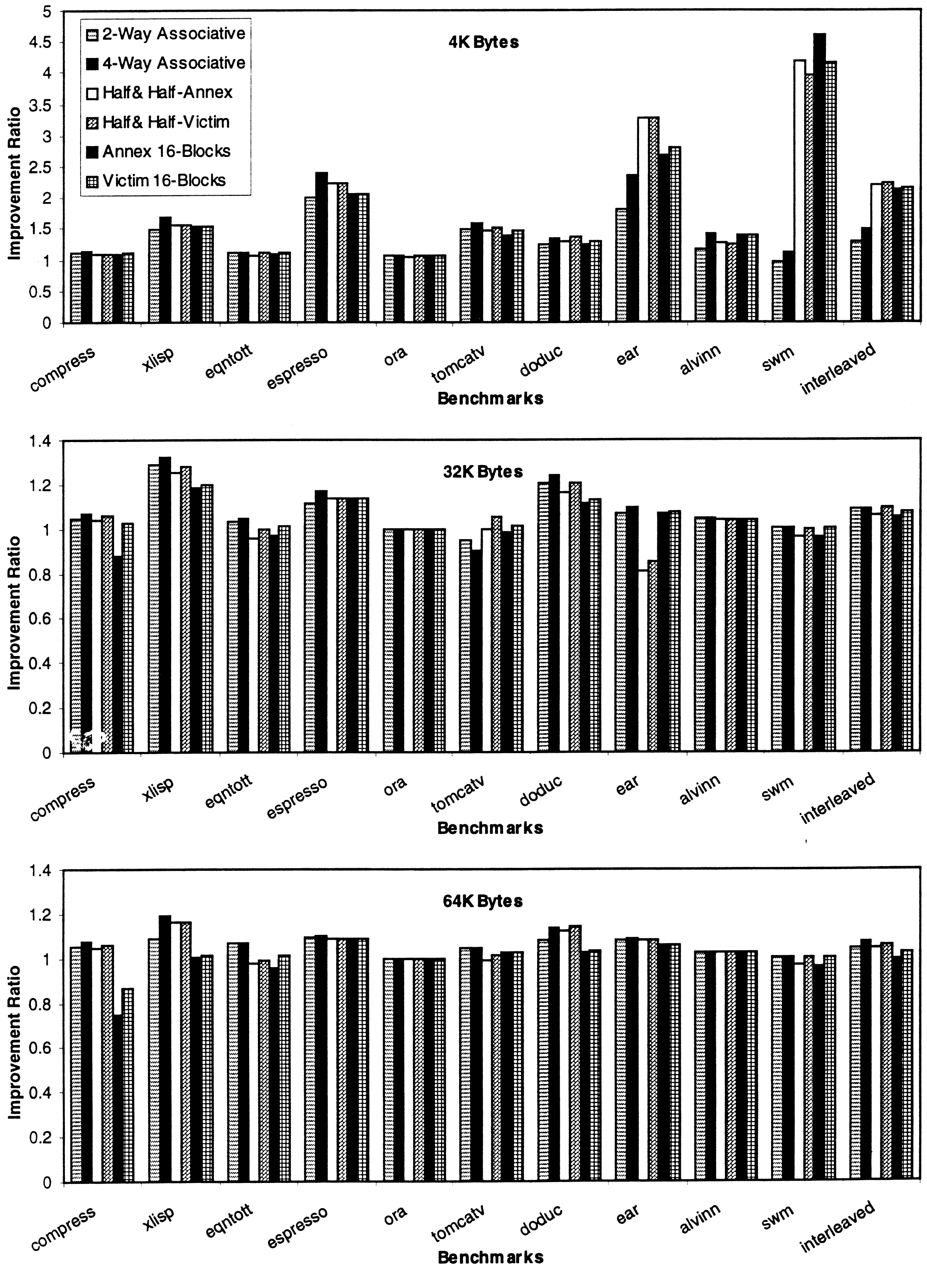

Fig. 5. Improvement in data access time for annex and victim caches compared to conventional caches.

set-associative caches. One benchmark in which annex has a noticeably poor performance is *compress*. 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 improvement in data access time by using annex caches, victim caches, half-and-half caches with annex caching policy, and half-and-half caches with victim caching policy, in comparison to a direct-mapped cache. The performance of 2- and 4-way set associative caches are also presented for reference. Annex caches are significantly better than conventional direct-mapped and 2-way set-associative caches and in most benchmarks, even better than 4-way set associative caches. However, annex caches do not display any

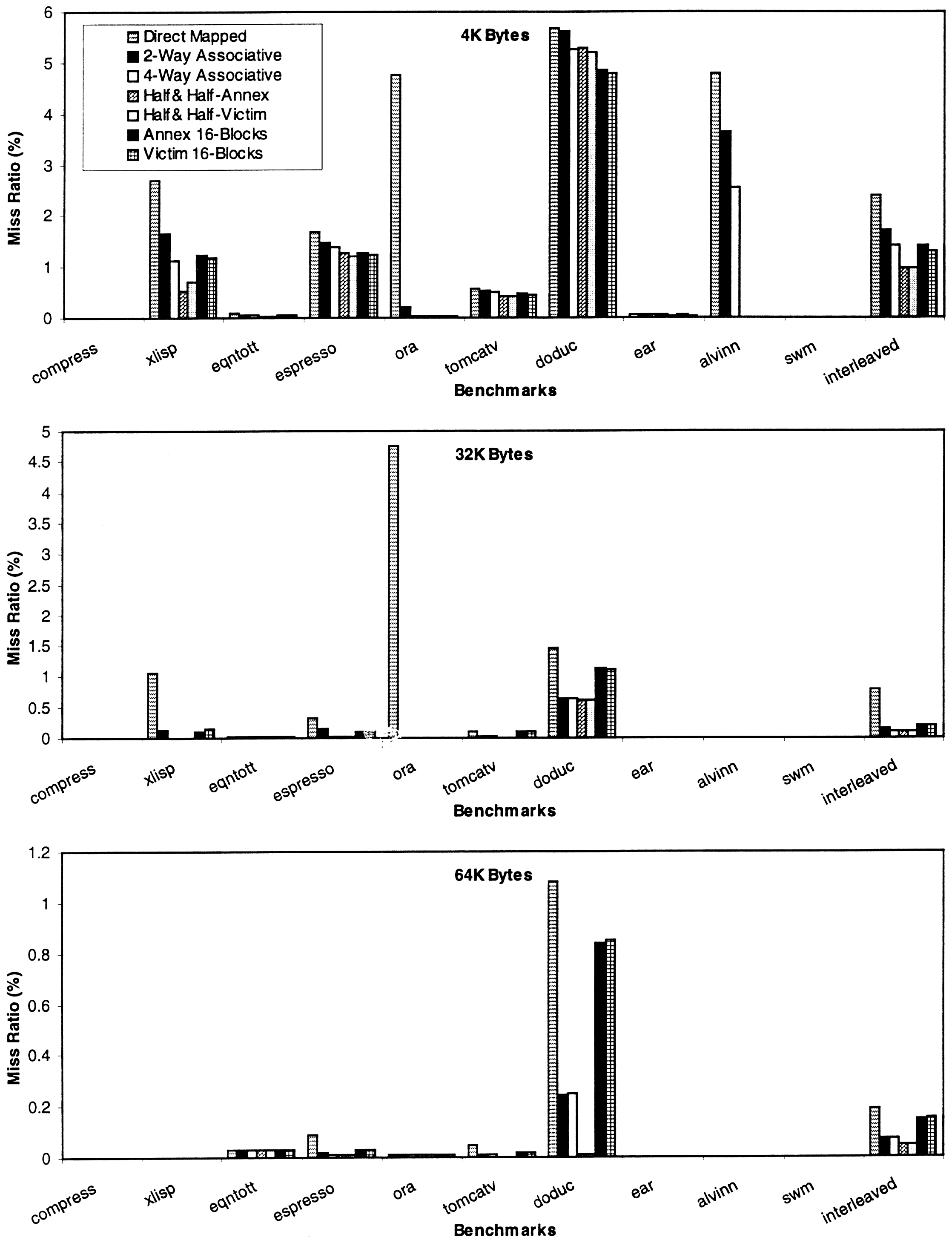

Fig. 6. Aggregate miss-rates in conventional, annex and victim caches for instruction traces.

dramatic improvement over victim caches as in the synthetic sequences.

Fig. 6 illustrates the miss ratios and Fig. 7 illustrates the improvement in access time for instruction traces. The annex/victim cache size is 16 blocks. Annex caching performs better than conventional direct-mapped, 2-way and in most cases even 4-way set associative caches. The significance of annex caching is greater at lower cache sizes.

The aggregate improvement from all the benchmarks are summarized in Table 4 for instruction and data traces. In this table, we also present results from annex and victim caches of size 4 blocks. Annex or victim caches of 4 block size are not seen to out-perform set-associative caches. However,

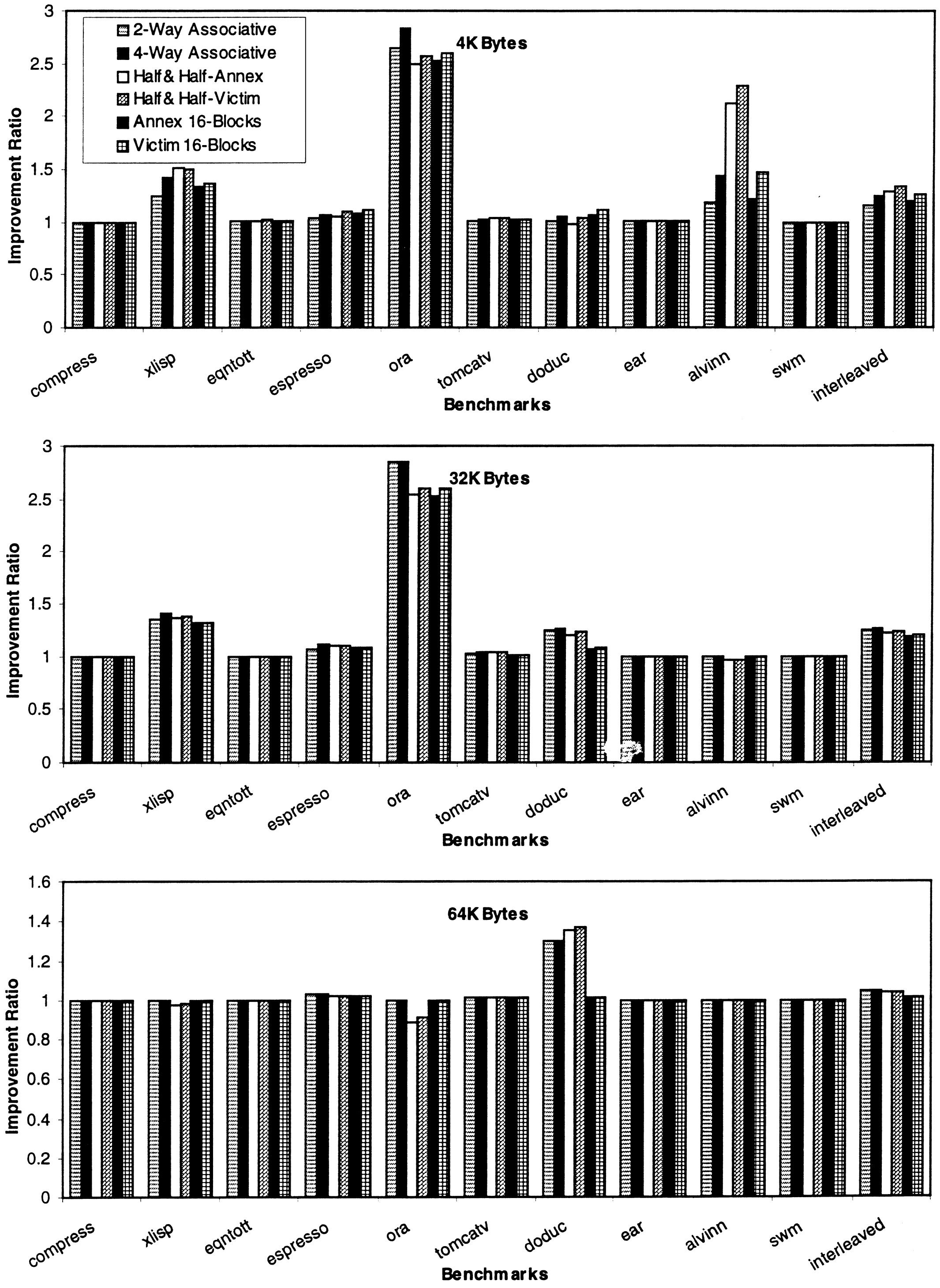

Fig. 7. Improvement in instruction access time for annex and victim caches compared to conventional caches.

one observation is that in instruction caching, the 4-block annex cache is better than the 4-block victim cache. In data caches, there really is no clear winner between the annex and victim algorithms either in the annex/victim or the half-and-half case. Half-and-half caching is a clear winner in the case of instructions and winner with no significant advantage in the case of data. While the improvement of annex caching over victim caching is not particularly striking, it is important to be aware of the performance improvement that can be obtained by combining cache exclusion with victim caching.

The major shortcoming of the annex cache is when its size is small. If the annex cache is very small, it does not allow references to go to the main cache. Many references

Table 4
Aggregate improvement over all benchmarks

	Configurations	4 kbytes	32 kbytes	64 kbytes
Data	Annex (16 blocks)	1.839	1.037	0.999
	Victim (16 blocks)	1.830	1.064	1.014
	2-Way set-associative	1.337	1.077	1.054
	4-Way set-associative	1.519	1.088	1.073
	Half-and-half annex	1.877	1.039	1.046
	Half-and-half victim	1.880	1.065	1.058
	Annex (4 blocks)	1.279	0.914	0.89
	Victim (4 blocks)	1.404	1.043	1.02
Instr.	Annex (16 blocks)	1.225	1.200	1.006
	Victim (16 blocks)	1.271	1.210	1.006
	2-Way set-associative	1.211	1.255	1.036
	4-Way set-associative	1.284	1.267	1.035
	Half-and-half annex	1.322	1.220	1.026
	Half-and-half victim	1.353	1.233	1.032
	Annex (4 blocks)	1.053	1.057	1.0
	Victim (4 blocks)	1.038	1.027	1.0

get knocked out from the annex cache itself due to conflicts within the annex. This problem explains the deterioration in performance obtained with 4-block annex caches at cache sizes of 32 and 64 kbytes for data. The problem reduces when the annex size is increased to 16 blocks. The most common size for cache assists is 16 blocks and hence more attention should be paid to the performance of the cache assists which are 16 blocks large.

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 annex cache cache combines the features of Jouppi's victim caches [1] and McFarling's cache exclusion schemes [8]. The performance of the proposed annex cache was evaluated using trace-driven simulation, for several programs in the SPEC 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 set-associative caches and cache assists such as the victim cache and half-and-half cache. In past research, McFarling [8] 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 [8] or other research on selective caching [10,11]. The quantitative comparison that is presented in this paper illustrates that annex caches are significantly better than conventional caches and comparable to victim caches.

Acknowledgements

This work was supported in part by the United States National Science Foundation under grants CCR-9796098 and EIA-9807112.

References

[1] 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.

[2] A. Agarwal, J. Hennessy, M. Horowitz, Cache performance of operating systems and multiprogramming, ACM Transactions on Computer Systems 6 (4) (1988) 393–431.

[3] A. Agarwal, S. Pudar, Column-associative caches: a technique for reducing the miss-rate of direct-mapped caches, International Symposium on Computer Architecture, 1993, pp. 179–190.

[4] J.H. Chang, H. Chao, K. So, Cache design of a sub-micron CMOS system/370, Proceedings of the 14th Annual International Symposium on Computer Architecture, Pittsburgh, PA, June 1987, pp. 208–213.

[5] K.B. Theobald, H.H.J. Hum, G.R. Gao, A design framework for hybrid-access caches, Proceedings of the First High Performance Computer Architecture Symposium, Raleigh, NC, January 1995, pp. 144–153.

[6] N. Topham, A. Gonzalez, J. Gonzalez, The design and performance of a conflict-avoiding cache, Proceedings of MICRO-30, Raleigh, NC, December 1997, pp. 71–80.

[7] S.J. Walsh, J.A. Board, Pollution control caching, Proceedings of the 1995 International Conference on Computer Design.

[8] S. McFarling, Cache replacement with dynamic exclusion, Proceedings of International Symposium on Computer Architecture, May 1992, pp. 191–200.

[9] A. Gonzalez, C. Aliagas, 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] C.-H. Chi, 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, June 1989, vol. 24, pp. 344–355.

[11] S.G. Abraham, R.S. Sugumar, B.R. Rau, R. Gupta, Predictability of load/store instruction latencies, Proceedings of MICRO-26, 1993, pp. 139–152.

[12] B. Bershad, D. Lee, T. Romer, B. Chen, Avoiding conflict misses dynamically in large direct-mapped caches, Proceedings of ASPLOS-VI, pp. 158–170.

[13] O. Temam, N. Drach, Software assistance for data caches, Proceedings of the High Performance Computer Architecture Symposium, January 1995, pp. 154–163.

[14] N.P. Jouppi, S.J.E. Wilton, Trade-offs in two-level on-chip caching, Proceedings of the 21st International Symposium on Computer Architecture, Chicago, IL, April, 1994, pp. 34–45.

[15] J.-L. Baer, W.-H. Wang, On the inclusion properties for multi-level cache hierarchies, *The 15th Annual Symposium on Computer Architecture*, IEEE Computer Society Press, Silver Spring, MD, 1988 pp. 73–80.

[16] UMIPS-V Reference Manual, MIPS Computer Systems, Sunnyvale, CA, 1990.

[17] M.D. Smith, Tracing with pixie, Technical Report No. CSL-TR-91-497, Computer Systems Laboratory, Stanford University, Stanford, CA.

Dr Lizy Kurian John is an Assistant Professor in the Department of Electrical and Computer Engineering, at the University of Texas at Austin, since September 1996. She received her PhD in Computer Engineering from the Pennsylvania State University in 1993. She was on the faculty of the Computer Science and Engineering department at the university of South Florida from 1993 to 1996. Her research interests include high performance processor and memory architectures, workload characterization and program behavior studies, compiler optimization for high performance processors, etc. Her research is supported by the National Science Foundation, the State of Texas Advanced Technology program, DELL Computer Corporation, IBM, AMD, Intel and Microsoft Corporations. She is recipient of an NSF CAREER award and a Junior Faculty Enhancement Award from Oak Ridge Associated Universities. She is a member of IEEE and its Computer Society and ACM and ACM SIGARCH. She is also a member of Eta Kappa Nu, Tau Beta Pi and Phi Kappa Phi.

Tao Li received his MS in computer science from Beijing Institute of Data Processing Technology, PR of China in 1996 and his BSE in Computer Science and Engineering from Northwestern Polytechnic University, PR of China in 1993, respectively. He is a graduate research assistant at the Laboratory for Computer Architecture, Department of Electrical and Computer Engineering, University of Texas at Austin since September 1998. His research interests include execution-driven simulation of computer architecture, cache and memory system design for shared memory multiprocessors.

Akila Subramanian received her masters in Computer Applications from the University of Madras, India. She was a Masters Degree student at the Computer Science and Engineering Department at the University of South Florida when this research was carried out.