Allocation By Conflict: A Simple, Effective Cache Management Scheme

Edward S. Tam, Gary S. Tyson, and Edward S. Davidson
Advanced Computer Architecture Laboratory
The University of Michigan
{estam,tyson,davidson}@eecs.umich.edu

Abstract

Many schemes have been proposed that incorporate an auxiliary buffer to improve the performance of a given size cache. One of the most thoroughly evaluated of these schemes, Victim caching, aims to reduce the impact of conflict misses in direct-mapped caches. While Victim has shown large performance benefits, its competitive advantage is limited to direct-mapped caches, whereas today's caches are increasingly associative. Furthermore, it requires a costly data path for swaps and saves between the cache and the buffer.

Several other schemes attempt to obtain the performance improvements of Victim, but across a wide range of associativities and without the costly data path for swaps and saves. While these schemes have been shown to perform well overall, their performance still lags that of Victim when the main cache is direct-mapped. Furthermore, they also require costly hardware support, but in the form of history tables for maintaining allocation decision information.

This paper introduces a new cache management scheme, Allocation By Conflict (ABC), which generally outperforms both Victim and the history-based schemes. Furthermore, ABC has the lowest hardware requirements of any proposed scheme — only a single additional bit per block in the main cache is required to maintain the information required by the allocation decision process, and no swap-save data path is needed.

1 Introduction

The Victim cache [1] was proposed as a method to reduce the impact of conflict misses in direct-mapped cache structures. While the Victim scheme performs well, it requires an auxiliary cache (buffer) to hold victims (replaced lines from the original (main) cache) and a data path between the two caches. Furthermore, the Victim scheme is targeted to direct-mapped caches, and its performance benefits degrade as the associativity of the main cache increases.

Several other schemes have been proposed which, like Victim, make use of a second cache placed in parallel with the main cache, but unlike Victim, attempt to achieve their performance gains without using a costly inter-cache data path. The best performing of these *multilateral* [2] schemes are NTS [3][4] and PCS [4], which make their allocation decisions between the two caches based on the past tour[1] usage of incoming blocks. A multilateral cache consists of two data stores with disjoint contents; the larger data store (or the "main" cache) is referred to as the A cache, while the smaller data store (or the "buffer") is referred to as the B cache. However, instead

1. A cache block tour is the time between an allocation of the block in the cache and its subsequent eviction. A cache block may have many tours through the cache during program execution.

of using an inter-cache data path, these schemes make use of a history table to contain the information necessary to make informed allocation decisions, so as to avoid swapping and saving blocks between A and B. NTS and PCS, however, generally fail to attain the performance gains provided by a Victim scheme when the main cache is direct-mapped, though contrary to Victim, their performance improves as associativity increases.

We have developed a new multilateral allocation scheme, Allocation By Conflict (ABC), in the hopes of outperforming Victim when A cache associativity is low, competing comparably with or outperforming NTS and other proposed reuse-based allocation schemes as associativity increases, and doing so with very low hardware complexity. ABC decides where to allocate blocks (into A or B) and which block from that cache to replace, as opposed to Victim's management (allocating into the A cache and saving victims into B). ABC makes its allocation decisions based on the current usage of blocks, not past tour performance. Put simply, ABC allocates a block to the A cache if the LRU element of its set in A has not been reaccessed since the last conflict to that set; otherwise, the block is allocated to the B cache. ABC's performance gains derive from its ability to reduce the impact of conflict misses and also its ability to remove blocks from the cache that are felt to be less useful than LRU elements. Furthermore, ABC makes its management decisions using only a single additional bit per block in cache A and no data path between the caches.

In this paper we discuss the ABC scheme operation in detail and compare the performance of ABC to Victim [1], NTS [3], and PCS [4]. To provide a performance baseline for multilateral structures of a given size and configuration, we also compare to and evaluate the performance of a Random allocation scheme. We find that the ABC scheme performs best of the reuse-based allocation strategies and ABC even performs comparably to Victim when A cache associativity is low. Interestingly, the Random scheme's performance is also good for such configurations, despite its lack of "intelligent" decision making. As the associativity of the A cache increases, the reuse-based schemes (NTS, PCS, and ABC) all continue to sustain their performance gains, while the gains offered by Random and Victim degrade.

In Section 2 we review previously proposed cache allocation schemes. In Section 3 we present our new cache management scheme, Allocation By Conflict, and detail our simulation environment in Section 4. We present the results and analysis of our experiments in Section 5 and Section 6, respectively. We conclude the paper in Section 7.

2 Previously proposed allocation schemes

Several schemes have been proposed for managing multilateral cache structures, in particular, Dual [5], NTS [3][4], MAT [6], and PCS [4]. Of these, NTS and PCS were found to perform best overall [4]. We compare the performance of ABC to NTS, PCS, and a Victim scheme [1], which is effectively a multilateral cache structure if swap penalties are ignored (set to 0). Victim with 0 swap penalty provides an upper bound on the performance of an implementable Victim scheme.

In this section we describe the operation of the Victim, NTS, and PCS schemes. We focus our discussion on multilateral first level (L1) caches for the following evaluations.

2.1 The Victim scheme

The Victim cache, based on the scheme proposed in [1], is a multilateral design where the B cache is managed as a victim buffer. The victim buffer is used to hold recently evicted items from the A cache in the hopes of keeping them there until their next access. The victim buffer has a ben-

eficial impact on performance when the A cache is direct-mapped and many conflicts occur, as much of the recently evicted data from the A cache will be available for fast access from the victim buffer (B cache).

On a memory access, desired data found in the A cache is returned to the processor the next cycle. If the desired data is found in the B cache, the desired block must first be "swapped" into the A cache — the block from the B cache is placed in the A cache and the block it evicts from A is placed in the B cache. Depending upon the amount of hardware dedicated to handling these swaps, a hit to the B cache may require varying amounts of time.

On a miss, the block entering the L1 cache structure from the next level of memory is placed in the A cache. A block evicted from A as the result of the new block's arrival is placed (saved) in the B cache; blocks evicted from the B cache return to the next level of memory. The Victim cache manages the cache state passively — recently evicted blocks are always saved in the B cache and any referenced block not found in A is moved to A.

Many previous evaluations have been performed on the Victim cache scheme, and several improvements to the basic Victim scheme have been proposed. Selective Victim Caching [7] was proposed as a way to reduce the number of blocks that are saved and swapped between the caches in order to reduce the performance impact of data movement while also increasing the performance of the scheme by selectively maintaining blocks in L1. Prediction Caches [8] were proposed as a way to combine streaming buffers [1] with Victim buffers and also to perform selective retention of blocks in L1. While both these schemes do improve upon the performance of the basic Victim scheme, each requires additional hardware beyond the inter-cache data path necessary for the basic Victim scheme. As we are focusing this paper on attaining the highest performance with the lowest cost, we use the Victim scheme as the performance baseline for Victim-like cache schemes, recognizing the fact that somewhat higher performance may be obtained using these latter proposed improvements, albeit at higher hardware costs.

2.2 The NTS scheme

The NTS (nontemporal streaming) cache [3] is a location-sensitive cache management scheme that uses hardware to dynamically partition cache blocks into two groups, temporal (T) and nontemporal (NT), based on their reuse behavior during a past tour. A block is considered NT if during a tour in L1, no word in that block is reused. Blocks classified as NT are subsequently allocated in the B cache; all other blocks (those marked T and those for which no prior information is available) are handled in the A cache. Data placement is decided by using reuse information that is associated with the effective address of the requested block. The effectiveness of NTS in reducing the miss ratio, memory traffic, and the average access penalty has been demonstrated primarily with mostly numeric programs.

The NTS cache, using the model in [4], operates as follows: On a memory access, if the desired data is found in either A or B, the data is returned to the processor and the block remains in the cache in which it is found. On a miss, the block entering L1 is checked to see if it has an entry in the Detection Unit (DU). The DU contains temporality information about blocks recently evicted from L1. Each entry of the DU describes one block and contains a block address (for matching) and a T/NT bit (to indicate the temporality of its most recent tour). On eviction, a block is checked to see if it exhibited temporal reuse (i.e. if some word in the block was referenced at least twice during this just-completed tour in the L1 cache structure), and its T/NT bit is set accordingly in the DU. On a miss, if the miss block address matches an entry in the DU, the T/NT

bit of that entry is checked and the block is placed in A if it indicates T, and B if not. If no matching DU entry is found, the miss block is assumed to be temporal and placed in A.

2.3 The PCS scheme

The PCS (program counter selective) cache [4] is a multilateral cache design that evolved from the CNA cache scheme [9]. The PCS cache decides on the data placement of a block based on the program counter value of the memory instruction causing the current miss, rather than on the effective address of the block as in the NTS cache. Thus, in PCS, the tour performance of blocks recently brought to cache by this memory accessing instruction, rather than the recent tour performance of the current block being allocated, is used to determine the placement of this block. The performance of PCS is best for programs in which the reference behavior of a given datum is well-correlated with the memory-referencing instruction that brings the block to cache.

The PCS cache structure modeled is similar to the NTS cache. The DU is indexed by the memory accessing instruction's program counter, but is updated in a manner similar to the NTS scheme. When a block is replaced, the temporality bit of the DU entry associated with the PC of the memory accessing instruction that brought the block to cache at the beginning of this tour is set according to the block's reuse characteristics during this just-completed tour of the cache. When an L1 miss occurs, the loaded block is placed in B if the accessing instruction's PC hits in the DU and the prediction bit indicates NT; otherwise, the block is placed in A.

3 Allocation By Conflict and Random allocation

Despite the performance benefits of the previously proposed multilateral cache schemes, their hardware complexity is high (data path between caches for Victim and history detection and maintenance mechanisms for NTS and PCS) and their performance varies as the configurations of the caches change. The Allocation By Conflict (ABC) scheme makes allocation decisions based on the current usage of data in the cache. In particular, ABC focuses on reducing the impact of conflict misses by prolonging the tour lengths of LRU, but still actively referenced blocks in the A cache. When a miss occurs, ABC focuses on the *conflict block*, i.e. the LRU block in the set of A that the miss block maps to — namely, the block that the miss block would replace if it were allocated into A. For conciseness, we refer to the set of A to which the miss block maps as the *A-set* and the set of B to which the miss block maps as the *B-set*. Whenever a miss block is allocated to B we say that a *CNR* (Conflict with No Replacement) has occurred in its A-set; note that each miss has an associated conflict block, but may or may not result in a CNR to its A-set.

3.1 ABC overview

In the ABC scheme, the LRU element of a miss block's A-set, α, is replaced unless it has been referenced since the last CNR to this A-set. If α has not yet experienced a CNR to its A-set during this tour or if it has been accessed since the last CNR to its A-set, the incoming block is allocated to B, replacing the LRU element in its B-set. These decisions enable blocks allocated to A to survive at least one CNR to their set before being evicted. Furthermore, if a block is accessed at least once between successive CNRs to its A-set, it can remain in A indefinitely. If α has not been accessed since the last CNR to its A-set, it could have been replaced during that last CNR without harming the cache state from then until now; it is therefore considered to be a good candidate for replacement now. Thus, ABC makes active replacement decisions based on whether to

retain or replace the conflict block in A. Furthermore, it makes this decision as a function of the current tour behavior of the conflict block since the last CNR to its set. By contrast, NTS and PCS make their active allocation decisions based on the past tour characteristics of the miss block or miss instruction, respectively.

3.2 ABC implementation

Like NTS and PCS, there is no data path between the caches. Each block in cache A is augmented with a single bit, the Conflict bit (C). The `C=0` state indicates that either no CNRs to its A-set have occurred during the block's current tour or the block has been accessed since the last CNR to its A-set. If a miss occurs while the LRU block in the A-set (the conflict block) has `C=0`, then the miss block is allocated to B and the conflict block is thereby retained in A. The `C=1` state indicates that a CNR to its A-set has occurred during the current tour, but this block has not been reaccessed since the last such CNR. If a miss occurs while the conflict block has `C=1`, the conflict block in A is replaced by the miss block.

When a block *X* is brought into cache, if its target set in the A cache is not full, *X* is simply allocated to the larger A cache. When its target set in A is full, the ABC allocation mechanism is put into effect. Let α be the conflict block of the A-set, and β be the conflict block of the B-set. (Note that α and β may refer to different blocks in L1 at different points in time.) If the Conflict bit of α is equal to 0, *X* is allocated into B, causing β to be replaced. Whenever a miss block is allocated into B, the Conflict bit of every block in its A-set is set to 1 to indicate that a CNR has occurred for that set.

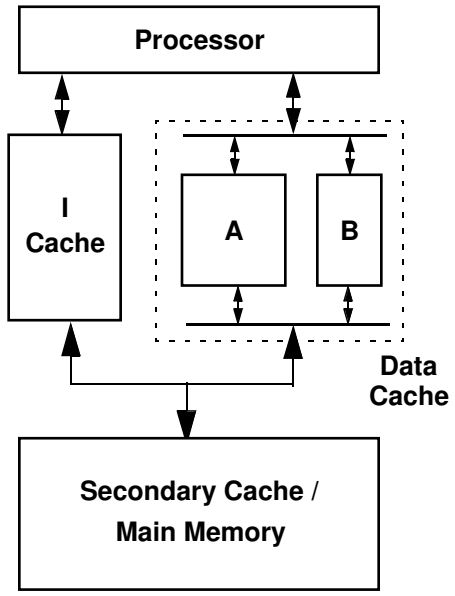
When a block in A is accessed (an A cache hit), its Conflict bit is reset to 0. If a subsequent miss block *Y* maps to that set in A and `C=0` for α, *Y* is placed in B and β is replaced. This decision is made because α has been used since the last CNR to the set has occurred (or, in the base case, that no CNR to this A-set has yet occurred during α's current tour), and α is thus deemed to have a good likelihood of being accessed again in the near future. When *Y* is placed in B, the Conflict bit of each A-set element is set to 1 (`C=1`), so that the A cache resident blocks must "prove" themselves by being referenced again in order to remain in L1 past the next CNR to this set.

If a subsequent miss block *Z* maps to that set in A and `C=1` for α, this indicates that α has not been used since *X* (or *Y*) was allocated in L1. Therefore, it is likely that α will not be used again soon and it could have been replaced by *X* or *Y* without harm to the cache state since the allocation of *X* (or *Y*). Thus, *Z* is allocated to A, replacing α. The Conflict bit of *Z* is set to 0 so that the new block has a chance to remain in A for at least a short period of time; otherwise, another conflicting block may miss and cause this new block to be evicted before it can be reused.

We have also evaluated two other schemes: 1) on CNR to an A-set, only the C bit of the LRU (tour-extended) block is set to 1, rather than all C bits in the A-set, and 2) each C bit is replaced with a 2-bit saturating counter [10]. However, neither of these schemes performed as well as the ABC scheme presented here.

3.3 Random allocation

We also evaluate the Random allocation scheme, which is used to set a performance baseline for multilateral caches of a given size and configuration. The Random allocation scheme has no data path between the A and B caches and uses LRU replacement in each cache; however, blocks are allocated to either the A or B caches by simply using a "virtual coin-flip" to decide whether the block is allocated to A or B. While this scheme uses no intelligence in its allocation decisions,



Fetch Mechanism	fetches up to 16 instructions in program order per cycle
Branch Predictor	perfect branch prediction
Issue Mechanism	out-of-order issue of up to 16 operations per cycle, 256 entry instruction re-order buffer (RUU), 128 entry load/store queue (LSQ); loads may execute when all prior store addresses are known
Functional Units	16 INT ALUs, 16 FP ALUs, 8 INT MULT/ DIV, 8 FP MULT/DIV, 8 L/S units
Functional Unit Latency (total/issue)	INT ALU:1/1, INT MULT:3/1, INT DIV:12/ 12, FP ALU:2/1, FP MULT:4/1, FP DIV:12/ 12, L/S:1/1
Instruction Cache	perfect cache, 1 cycle latency
Data Cache	Multilateral L1 (A and B), write-back, write-allocate, 32 byte lines, 1 cycle hit latency, 18 cycle miss latency, non-blocking, 8 memory ports

Figure 1: Processor and memory subsystem characteristics.

Random does still take advantage of the LRU replacement strategies of the A and B caches, and provides a remarkably good performance baseline for a multilateral cache of a given size and configuration.

Random requires a random number generator that provides a uniform distribution of outputs so that half of the incoming blocks are allocated to A, and the other half to B. This randomness can be obtained any number of ways, e.g. a random number generator, sampling the real-time clock, etc. This 50/50 split of allocations could be adjusted depending upon the configurations of the A and B caches used, but in the performance comparisons that follow, Random always utilizes the 50/50 split.

4 Simulation environment

The processor modeled in this study is a modification of the *sim-outorder* simulator in the SimpleScalar [11] toolset. The simulator performs out-of-order (OOO) issue, execution, and completion on a derivative of the MIPS instruction set architecture. A schematic diagram of the targeted processor and memory subsystem is shown in Figure 1, with a summary of the chosen parameters and architectural assumptions.

The memory subsystem, modeled by the *mlcache* multilateral cache simulator [12], consists of a separate instruction and data cache and a perfect secondary data cache or main memory. The instruction cache is perfect and responds in a single cycle. Both A and B caches are non-blocking with 32-byte lines and single cycle access times. A standard (single structure) data cache model would simply configure cache A to the desired parameters and set the B cache size to zero.

The L2 cache access latency is 18 cycles; a 256 bit bus between L1 and L2 has 32 bytes/cycle data bandwidth. L1 to L2 access is fully pipelined; a miss request can be sent on the L1-L2 bus every cycle for up to 100 pending requests. The L2 cache is modeled as a perfect cache in order to focus this study on the effectiveness of management strategies for the L1.

The cache schemes we chose to evaluate are: Victim, NTS, PCS, ABC, and Random, along with various single structure caches; the evaluated configurations are summarized in Table 1. NTS

	Single	Victim		ABC/NTS/PCS/Random	
Cache	A	A	B	A	B
Size	8/16K	8K	1K	8K	1K
Associativity	1,2,4,8/1,2,4	1	full	1	full
Replacement Policy	–,LRU,LRU,LRU/–,LRU,LRU	–	LRU	–	LRU
move time	–	0/2/4		–	
latency to next level	18	18	–	18	18

TABLE 1: Characteristics of the six configurations studied. Times/latencies are in cycles.

and PCS both make use of a 32-entry DU. We used the SPEC95 benchmark suite for our evaluations, running the test inputs. Each benchmark was run to completion or through the first 1.5 billion instructions.

5 Experimental results

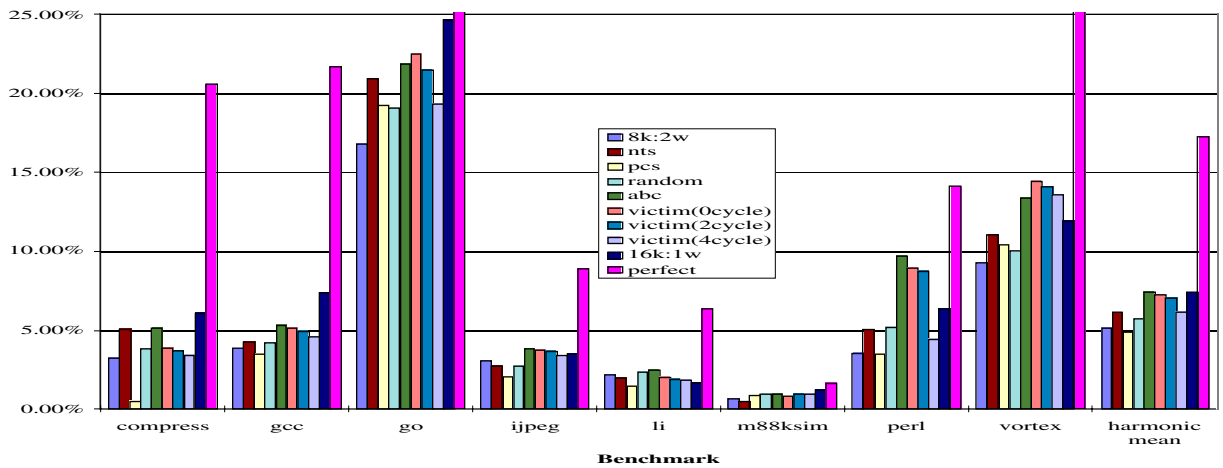
In this section, we present the performance of the direct-mapped A and fully-associative B cache multilateral cache configurations vs. single structure direct-mapped and 2-way associative caches of similar size (8K). We then present the performance of the multilateral schemes as the associativity of the A cache is increased from direct-mapped to 2- and 4-way set associative. We perform detailed analysis of each cache scheme's performance in Section 6.

5.1 Direct-mapped A cache performance

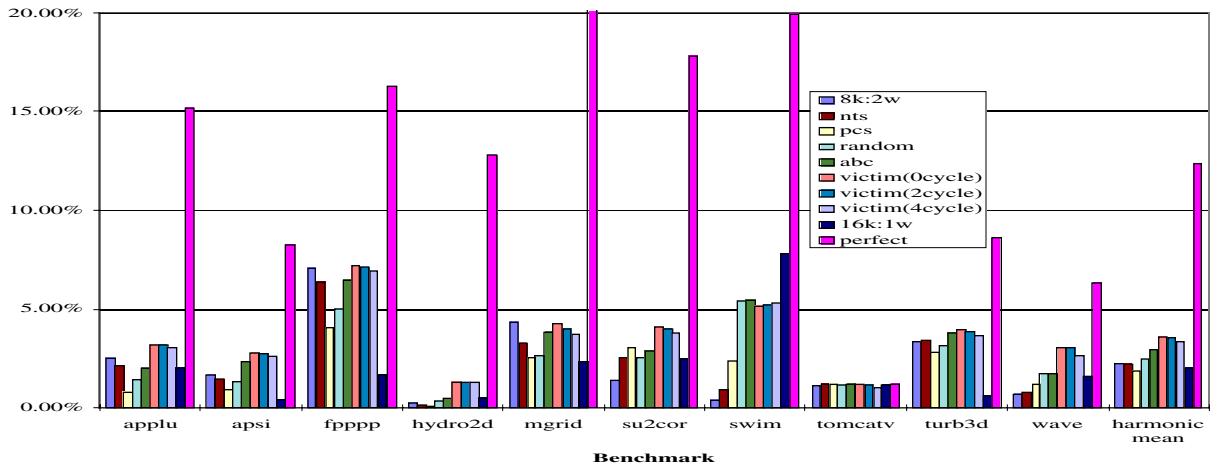
The speedup achieved by each scheme for each program is shown in Figure 2, where the single 8K direct-mapped (8k:1w) cache is taken as the base. Overall, the speedup obtained by using the multilateral cache schemes ranges from virtually none in **hydro2d** to ~22% in **go** with Victim (0 cycle swap latency). Clearly, some of the benchmarks tested do not benefit significantly from any of the improvements offered by the cache schemes evaluated, i.e. better management of the L1 data store by the multilateral schemes, increased associativity of a single cache (8k:2w), or a larger cache (16k:1w). In benchmarks where there is appreciable performance gain over the base cache, the multilateral schemes often perform as well as or better than either a higher-associative single cache or a larger direct-mapped cache. However, in **compress**, **gcc**, **go**, and **swim**, the benchmark's larger working sets do benefit most from the larger overall cache space provided by the 16K direct-mapped structure, although even for these benchmarks the multilateral schemes are able to obtain a significant part of the performance boost via their better management of the smaller multilateral cache.

Among the multilateral schemes, we see several trends that were exposed in earlier studies [2][4][12][13]. With a direct-mapped A cache, the NTS scheme generally outperforms the PCS scheme (in all benchmarks except **m88ksim**, **su2cor**, **swim**, and **wave**), and the Victim scheme generally outperforms both NTS and PCS (the exceptions being **compress** and **tomcatv**, where NTS is best). Even when a swap latency is incorporated for the Victim scheme (either 2 or 4 cycles, as opposed to the base 0 cycle (free) swap), Victim outperforms NTS and PCS. However, when the associativity of the A cache increases, the performance of Victim decreases relative to NTS and PCS, as we shall see in Section 5.2.

Interestingly, the Random allocation scheme also performs very well, equalling or beating both NTS and PCS in nearly half of the benchmarks (**ijpeg**, **li**, **m88ksim**, **perl**, **hydro2d**, **swim**, **tomcatv**, and **wave**) and in the harmonic mean of SPECfp, although NTS does outperform Ran-



i) SPECint95



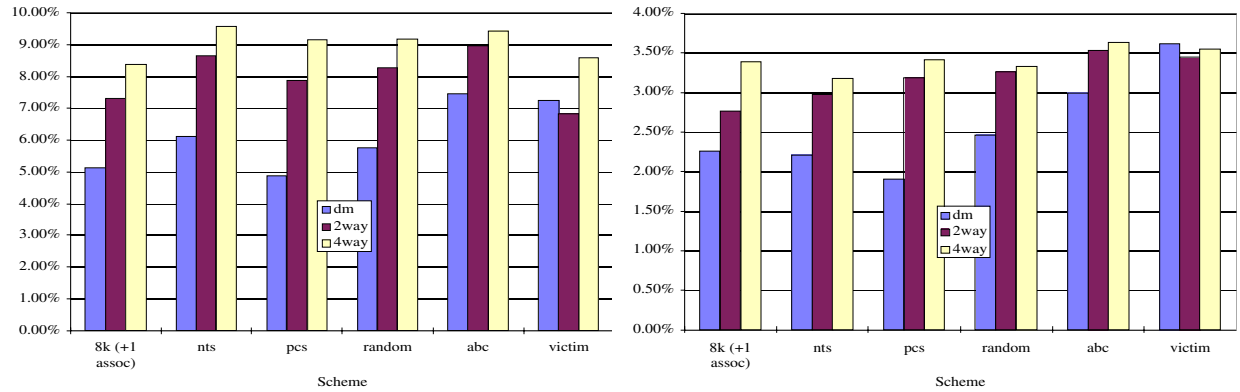
ii) SPECfp95

Figure 2: Overall execution speedup for the six evaluated cache schemes, relative to a single direct-mapped 8K cache (8k:1w). The three Victim results show the effect of increasing the swap latency between the caches. Perfect is the performance of a perfect cache. The harmonic mean speedup over the SPECint and SPECfp benchmarks is also shown. The Perfect speedup for go is 41.86%, vortex 33.8%, mgrid 20.4%, and swim 20%.

dom in the harmonic mean of SPECint. Random's performance is intriguing since, unlike NTS and PCS, it does not take into account any reuse information when making its allocation decisions.

ABC is the top performing multilateral scheme among those that do not allow data movement between the subcaches (i.e. among NTS, PCS, Random, and ABC), and even outperforms Victim (even with 0 cycle swap latency) in all of the integer benchmarks except **go** and **vortex**. In the floating point benchmarks, ABC's performance lags Victim's, though as the swap latency increases, Victim's performance drops and becomes more comparable to ABC's performance.

Looking at the harmonic mean speedups, both the ABC and Victim caches equal (SPECint) or outperform (SPECfp) an associative cache of similar size (8k:2w) and a single structure cache nearly twice as large (16k:1w). In the case of the ABC cache, these performance gains are attained using very little additional hardware. The best harmonic mean speedups over the base 8k:1w cache are 7.45% (ABC) over the integer benchmarks and a more modest 3.61% (Victim with 0



i) SPECint95 **ii) SPECfp95**

Figure 3: Harmonic mean speedups of the multilateral cache schemes over 8k:1w as associativity increases. The effects of increasing associativity on a single structure cache are also shown under 8k (+1 assoc) – the dm **bar indicates 8k:2w performance,** 2way **shows 8k:4w performance, and** 4way **shows 8k:8w performance. Victim has 0 cycle swap latency.**

cycle swap latency) over the floating point benchmarks.

The results scale well as larger cache sizes are used (and the working sets of the benchmarks do not fit in the cache). The performance trends for (16+2)K multilateral cache structures (i.e. 16K direct-mapped A cache plus a 2K fully-associative B cache) vs. 16K direct-mapped, 16K 2-way, and 32K direct-mapped single structure caches mirror those found for the (8+1)K multilateral caches vs. a single structure 8K direct-mapped cache: ABC is the best performing scheme among the SPECint benchmarks and Victim performs best overall for the SPECfp benchmarks [10]. The overall performance gains over the base cache, however, are generally smaller when the caches are larger, as the base cache is able to contain more of the active working set.

5.2 Associative A cache performance

The harmonic mean speedups achieved by each scheme as the associativity of the A cache is increased to 2- and 4-way is shown in Figure 3. The base configuration in each case is the single 8k:1w cache, so that we may compare the performance of the associative A cache multilateral caches to the direct-mapped A cache multilateral caches shown in Figure 2.

[NOTE TO REVIEWERS: Detailed speedup figures for the 2- and 4-way associative multilateral caches are included in the appendix.]

We see, from Figure 2 and Figure 3, that as the associativity of the A cache has increased to 2-way, the NTS and PCS speedups have increased and the Victim scheme's speedup has dropped relative to the corresponding base cache, concurring with the results found in [4]. Over the integer benchmarks, the performance of NTS is on par with the best performing scheme, ABC, and beats or equals PCS. Over the floating point benchmarks, ABC is best, followed by Victim, Random, PCS, and finally NTS.

The Victim scheme evaluated in these experiments incurs a 0 cycle latency for swaps, and yet its performance still trails that of ABC, and is often outperformed by NTS and PCS. If a nonzero swap latency is incurred (as it likely would be, in any real implementation of Victim), Victim's performance would drop further.

The Random allocation scheme, interestingly, still performs quite competitively, though its overall performance is exceeded by NTS in addition to ABC (except in harmonic mean SPECfp).

Fetch Mechanism	fetches up to 4 instructions in program order per cycle
Instruction Cache	perfect cache, 1 cycle latency
Branch Predictor	Bi-Mode branch predictor
Issue Mechanism	out-of-order issue of up to 4 operations per cycle, 16 entry re-order buffer (RUU), 8 entry load/store queue (LSQ)
Functional Units	4 integer ALUs, 4 FP ALUs, 1integer MULT/DIV, 1FP MULT/DIV, 2 L/S units
Functional Unit Latency (total/issue)	integer ALU:1/1, integer MULT:3/1, integer DIV:12/12, FP adder:2/1, FP MULT:4/1, FP DIV:12/12. load/store:1/1
Data Cache	write-back, write-allocate, 32B lines, 2 read/write ports, non-blocking

TABLE 2: Characteristics of the less aggressive processor configuration.

As the associativity of the A cache is increased to 4-way, the performance of all of the multilateral schemes begin to converge, with the NTS, PCS, and ABC schemes exhibiting comparable harmonic mean speedups over the base 8k:1w cache [10]. All of these schemes outperform an 8-way cache of the same size (8k:8w) and are even competitive with a larger 4-way cache (16k:4w). Random's performance is comparable, though slightly lower than any of the "intelligent" multilateral cache schemes. Victim's speedup with 2- or 4-way associative structures, even with a best case 0 cycle swap latency, is beaten by all the multilateral schemes over the integer benchmarks and is outperformed by ABC over the floating point benchmarks.

5.3 The effect of a less aggressive processor engine

The experiments presented thus far have modeled a processor with perfect branch prediction and an aggressive OOO processor engine. However, imperfect branch prediction can greatly affect overall performance, as well as the performance improvement garnered by the use of better managed cache structures. Also, if a sufficient number of instruction buffer entries in the RUU is available to allow a large number of instructions to be in flight at once, more of the effects of cache misses may be masked by performing unrelated useful work.

We performed simulations of the same cache configurations (with direct-mapped A and fully-associative B caches) used in Section 5.1, except that in place of a perfect branch predictor, we used the Bi-Mode branch predictor [14], and we scaled the processor engine down to an effective 4-wide superscalar machine; a description of the less aggressive processor is shown in Table 2. While other branch predictors may give slightly better performance for some of the benchmarks, the Bi-Mode branch predictor realistically represents the effect of one of today's best implementable branch prediction schemes on program execution times.

The performance speedups of each of the cache schemes over the base 8K direct-mapped cache are shown in Table 3. The relative performance of the different cache schemes mirrors that shown in Figure 2: ABC has the best overall performance over the SPECint benchmarks, and Victim's performance is best overall over the SPECfp benchmarks, with ABC a close second. The best harmonic speedups have increased as processor masking effects have been reduced via the imperfect branch prediction and less useful work can be found in the instruction buffers, going from 7.45% to 8.62% for ABC over SPECint and from 3.61% to 10.00% for Victim (2.99% to 9.57% for ABC) over SPECfp. Even as we increase the cache miss (memory) latency by four times to 72 cycles, the performance rating among the various schemes is very similar (though the absolute performance as measured by IPC decreases, as expected); ABC is the best over SPECint, while Victim (with 0 cycle swap latency) is best over SPECfp. Since the memory is four times

		18 cycle miss latency					72 cycle miss latency				
		NTS	PCS	Victim	ABC	Rand	NTS	PCS	Victim	ABC	Rand
SPECint95	**Compress**	3.27	2.13	3.49	**3.85**	3.12	10.32	7.09	11.48	**12.48**	10.32
	Gcc	5.18	4.67	6.60	**6.86**	5.51	14.53	12.98	18.82	**19.45**	15.44
	Go	13.42	11.96	**15.26**	14.75	12.62	30.47	26.99	**34.82**	33.70	28.74
	Ijpeg	4.26	3.61	5.04	**5.05**	4.20	14.35	12.08	16.99	**17.11**	14.09
	Li	5.31	4.02	5.33	**6.34**	5.10	15.06	11.25	15.43	**18.09**	14.53
	M88ksim	4.51	3.84	5.35	**5.45**	4.89	18.75	16.73	21.85	**22.26**	20.07
	Perl	6.83	6.39	7.90	**9.20**	7.90	22.15	20.24	**29.78**	29.77	25.73
	Vortex	17.14	15.99	**20.16**	19.28	15.79	32.03	29.77	**37.84**	35.78	29.51
	H-mean	7.29	6.39	8.38	8.62	7.23	19.26	16.68	22.73	23.08	19.41
SPECfp95	**Applu**	4.32	3.71	**4.78**	4.15	3.76	13.22	11.42	**14.58**	12.67	11.60
	Apsi	3.40	5.97	**6.82**	6.17	5.00	12.91	20.67	**22.77**	20.82	16.80
	Fpppp	15.99	7.93	15.43	**17.32**	12.90	34.92	15.88	31.98	**35.85**	26.68
	Hydro2d	1.72	0.16	2.08	**2.49**	1.52	3.79	1.05	4.39	**5.45**	3.64
	Mgrid	9.45	6.62	**9.80**	6.72	5.49	19.68	13.74	**20.37**	11.83	10.01
	Su2cor	3.15	3.78	**4.48**	4.06	3.54	8.28	10.10	**11.64**	10.94	9.60
	Swim	20.38	21.05	29.41	**30.37**	25.76	41.51	43.36	58.33	**59.20**	51.58
	Tomcatv	6.71	5.86	**6.94**	6.90	6.76	26.86	26.66	**27.78**	27.66	27.03
	Turb3d	3.70	3.06	**4.80**	4.16	4.05	9.64	8.28	**13.13**	11.22	11.10
	Wave	15.44	13.90	**21.27**	19.57	18.66	34.30	29.82	**47.74**	44.04	42.06
	H-mean	8.08	6.91	10.00	9.57	8.27	19.27	17.00	23.39	21.95	19.35

TABLE 3: Performance speedup (in %) over a single direct-mapped 8K cache for each scheme when a less aggressive processor (4-wide superscalar, with Bi-Mode branch prediction) is used and the miss latency is 18 cycles (left) and 72 cycles (right). H-Mean is the Harmonic mean speedup over the SPECint and SPECfp benchmarks. Best performing scheme for each benchmark (and Harmonic Mean) per miss latency is in bold. Victim has 0 cycle swap latency. Random (Rand) uses the 50/50 allocation ratio.

slower than in the other experiments, the effects of good cache usage (i.e. higher hit rates) has a greater impact on overall performance speedup — the best harmonic mean speedups over the SPECint and SPECfp benchmarks are now 23.08% (ABC) and 23.39% (Victim), respectively.

6 Performance analysis

In this section, we analyze the performance of the multilateral schemes running the SPEC95 benchmarks with the aggressive processor configuration (described in Section 4). We first evaluate the surprising performance of the Random scheme in Section 6.1, followed by some observations on the NTS and PCS schemes in Section 6.2. We then discuss the performance differences between ABC and Victim in Section 6.3.

6.1 Random allocation performance

The Random allocation scheme is interesting in that although it would typically serve as a baseline for comparison for multilateral structures of a given size and configuration, in these experiments, we found that in some cases the Random scheme performed better than either NTS

or PCS, particularly on SPECfp95 with a direct-mapped A cache. This performance is possible for several reasons: 1) Random allocates blocks to both A and B equally (for the 50/50 ratio), thereby distributing tours equally among both caches. While a given block may in fact benefit, i.e. experience longer tours, when allocated to the higher associative B cache or the larger size A cache, when the block is evicted from a sub-optimally allocated tour, it has a 50/50 chance of being allocated to its preferred cache structure for its next tour. As a result, chances are quite high that if a block is accessed frequently by the program, it will find the best place to remain in the multilateral L1. 2) Each of the caches is managed via LRU replacement, so blocks that are recently accessed have a high likelihood of remaining in the cache they are allocated to. Using the aid of LRU replacement, blocks that are the most used will stay in the A or B cache as long as they are accessed, and those that fall out via LRU have a chance of being allocated better during their next tour. Thus, despite not making any "intelligent" allocation decision for a miss block, the use of LRU and the even allocation split between the two caches gives a block a reasonably good chance of being found in its preferred cache within L1.

Over all the benchmarks, in terms of harmonic mean speedup, the 50/50 ratio is the most consistent performer [10]. If certain workloads are targeted, the ratio of allocations in the Random scheme might be set to attain higher performance, but the 50/50 ratio appears to be best overall.

While the Random scheme's performance is reasonable, its behavior is not easily predictable across different benchmarks. Schemes whose allocation decisions are based on some current or past reuse information, like the ABC, NTS, and PCS schemes, are likely to be more easily understood and do indeed perform better overall as the cache configurations are varied. However, Random's good performance does cast some doubt on the benefits of relying on past tour reuse information for making allocation decisions. NTS and PCS actively attempt to place a miss block in L1 based on its predicted usage. However, they may mistakenly tag a block or referencing instruction as T or NT and repeatedly place correlated blocks suboptimally in A or B. Random, which uses no history information and makes a new choice for each tour, can recover from its mistakes more quickly. While the history-based mechanisms of NTS and PCS do appear to perform better as A cache associativity increases, when the A cache is direct-mapped, Random gives comparable performance to NTS and PCS. Furthermore, Random has slightly less hardware complexity (i.e. a random number generator rather than temporality detectors and a DU for storing reuse information).

6.2 ABC vs. NTS and PCS

While the NTS and PCS schemes make their block allocation predictions based on past usage of miss blocks (NTS) and other blocks associated with them (PCS), the ABC scheme determines whether to allocate the new miss block into A or B by looking only at the *current* usage of the blocks *in cache A* that conflict with the miss block. Using the current state of the cache to make placement decisions can actually result in a better performance improvement than using past tour usage behavior, as tour usage behavior may not be persistent [13]. Also, the ABC scheme makes its allocation decision based on whether to replace the conflict block in A, while the NTS/PCS schemes only look at the miss block's (NTS) or instruction's (PCS) temporality usage in previous tours. It is thus possible in NTS/PCS that many blocks are loaded into the A cache because they are marked T; if several of these blocks should indeed reside in the cache concurrently and the associativity of the A cache is not sufficient, blocks that will be accessed soon in the future will be kicked out. It would be better if all of these blocks could reside in the L1 at once, and the ABC

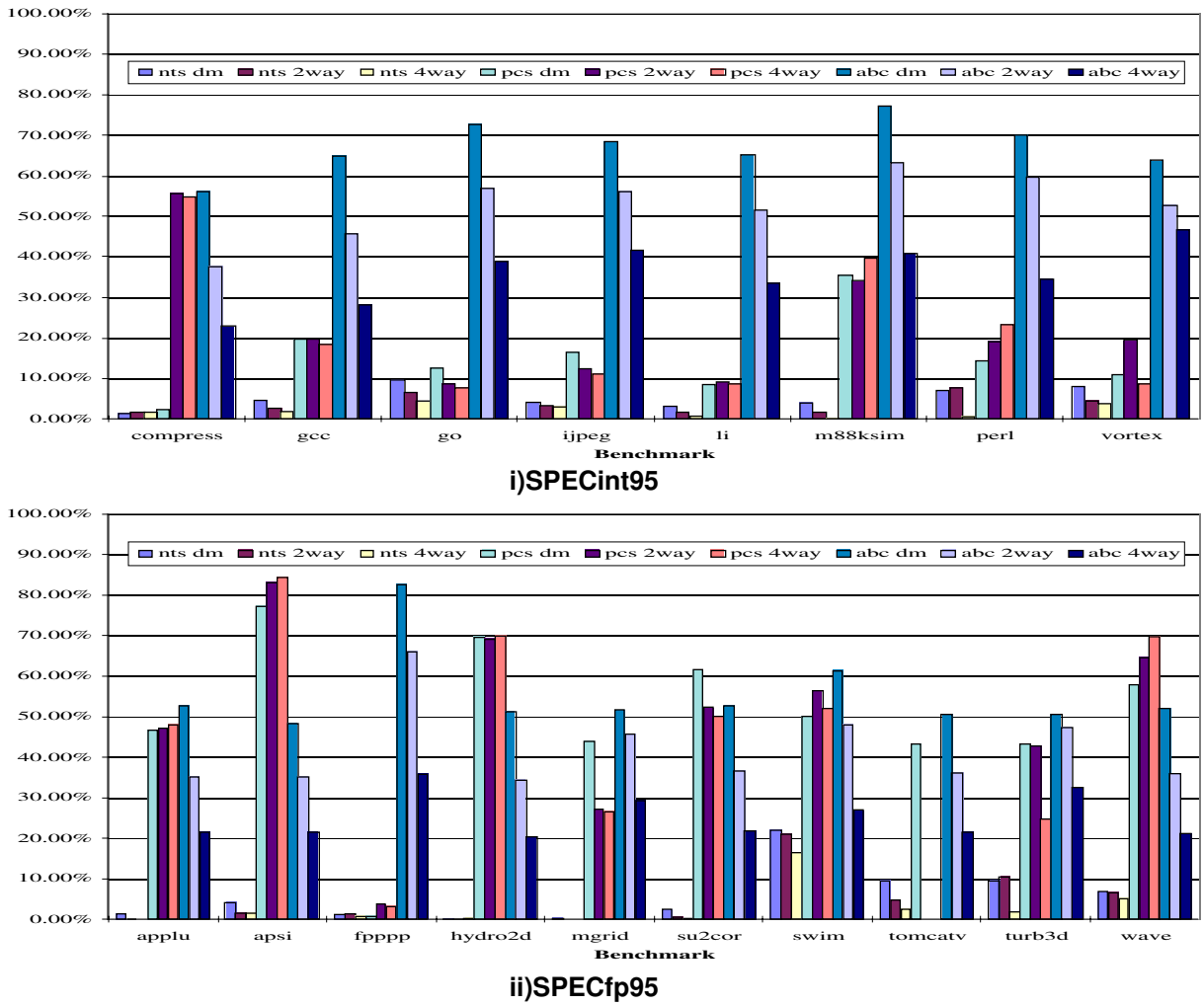


Figure 4: Fraction of time that tours are allocated to the B cache for ABC, NTS, and PCS.

scheme allows them to do so. Furthermore, ABC's allocation decision can be made using a simple logic circuit, while the NTS/PCS allocation decisions must be made via a table lookup with an associative compare.

We can see the difference in allocation decisions by looking at how often each cache scheme allocates blocks to A or B. Recall that blocks are allocated to B in NTS and PCS only if the block or instruction, respectively, exhibited NT behavior during its most recent block tour and that information is still available in the DU. Figure 4 shows the fraction of tours that are allocated to the B cache in the NTS, PCS, and ABC schemes for the SPEC95 benchmark suite. In general, the B cache is not used very much in either NTS or PCS for the SPECint benchmarks — less than 23% of all block tours under PCS and less than 10% under NTS are allocated to B. While the size of the B cache is indeed much smaller than the A cache, we see that for ABC and Random (which allocates to B 50% of the time), the B cache is used much more.

When the associativity of the A cache is low, too many conflicts occur and the DU is overrun by recently evicted block reuse information, so much so that little reuse information remains in the DU long enough for use in future block allocation predictions. As the associativity of the A cache increases, fewer conflicts occur, the DU is not stressed as greatly, and blocks that are

evicted from A that show nontemporal reuse are more likely to actually be nontemporal blocks; the longer a block stays in the cache, the more likely its reuse behavior is to reflect its optimal reuse characteristics [13]. As a result, the performance of NTS and PCS improves relative to Victim and Random as A cache associativity increases.

The ABC scheme more readily allocates blocks to B because its decisions are based solely on whether to extend the tour length of the conflict (LRU) block in the A cache. We see from Figure 4 that the fraction of blocks allocated to B decreases as the associativity of A increases. This phenomenon makes sense, as the number of conflicts typically decreases as A cache associativity increases. Furthermore, as A cache associativity increases, the likelihood of finding a conflict block in the A-set that has not been accessed since the last CNR to that set increases as well, resulting in a potential increase in the number of blocks allocated to A. Nevertheless, even when the A cache is 4-way associative, the ABC cache still allocates at least 20% of all its tours to the B cache, a higher percentage than either NTS or PCS allocate to B (except for PCS in SPECfp). However, as the reasons for allocating to A vs. B differ between NTS, PCS, and ABC, allocating more blocks to B does not guarantee better performance.

6.3 ABC vs. Victim

Inherently, the way both the ABC and Victim schemes attain their performance gains is similar: both reduce the impact of hot sets by providing dynamically-sized cache sets, and both achieve this by using the B cache as a buffer that may contain several blocks that map to the same set of the A cache. In both ABC and Victim, the allocation of some block to B occurs when a conflict occurs in cache A. However, beyond that, their approach to providing these dynamically-sized sets differs. Victim in effect may create a larger size cache set whenever a conflict occurs in a set of A — the new miss block is allocated to A and the replaced block, the victim of that allocation, is saved in the B cache. If the element replaced from B due to the victim's allocation maps to the same set as the new miss block, the dynamic size of that set remains constant. If the replaced block and the victim are from different sets, the dynamic size of the victim's set increases, and that of the replaced block's set decreases by one. The ABC scheme, on the other hand, may only create a larger size cache set when a conflict is found in A and the conflict block has yet to experience a CNR, or the conflict block has been accessed since the last CNR to that A-set. In this case, the conflict block is retained in A and the miss block is allocated into B; if the miss block and the replaced block map to different sets in A, the dynamic size of the miss block's set increases by one block and the dynamic size of the replaced block's set decreases by one. In all other cases, the miss block is allocated into A and all set sizes remain the same.

Reducing the dynamic size of cache sets is beneficial when few of the conflicting elements are soon reused or need to reside in the cache concurrently. If, for instance, Victim saves an element α in the B cache that is not reused before it is evicted from B, α need not have been saved and some other element in B could instead have remained in L1. In the ABC cache, conflict elements in A may remain in A, but to do so, they must prove that they are useful by being accessed at least once between CNRs to their A-set. The B cache is then used to buffer miss blocks whose conflict blocks qualify to remain in A; blocks that are replaced from A or B return to the next level of memory.

When the A cache is direct-mapped, the ABC scheme outperforms the Victim cache in all but two of the SPECint95 benchmarks (**go** and **vortex**). In general, Victim is saving too many useless elements into the B cache and is not using the B cache as effectively as the ABC scheme. ABC's

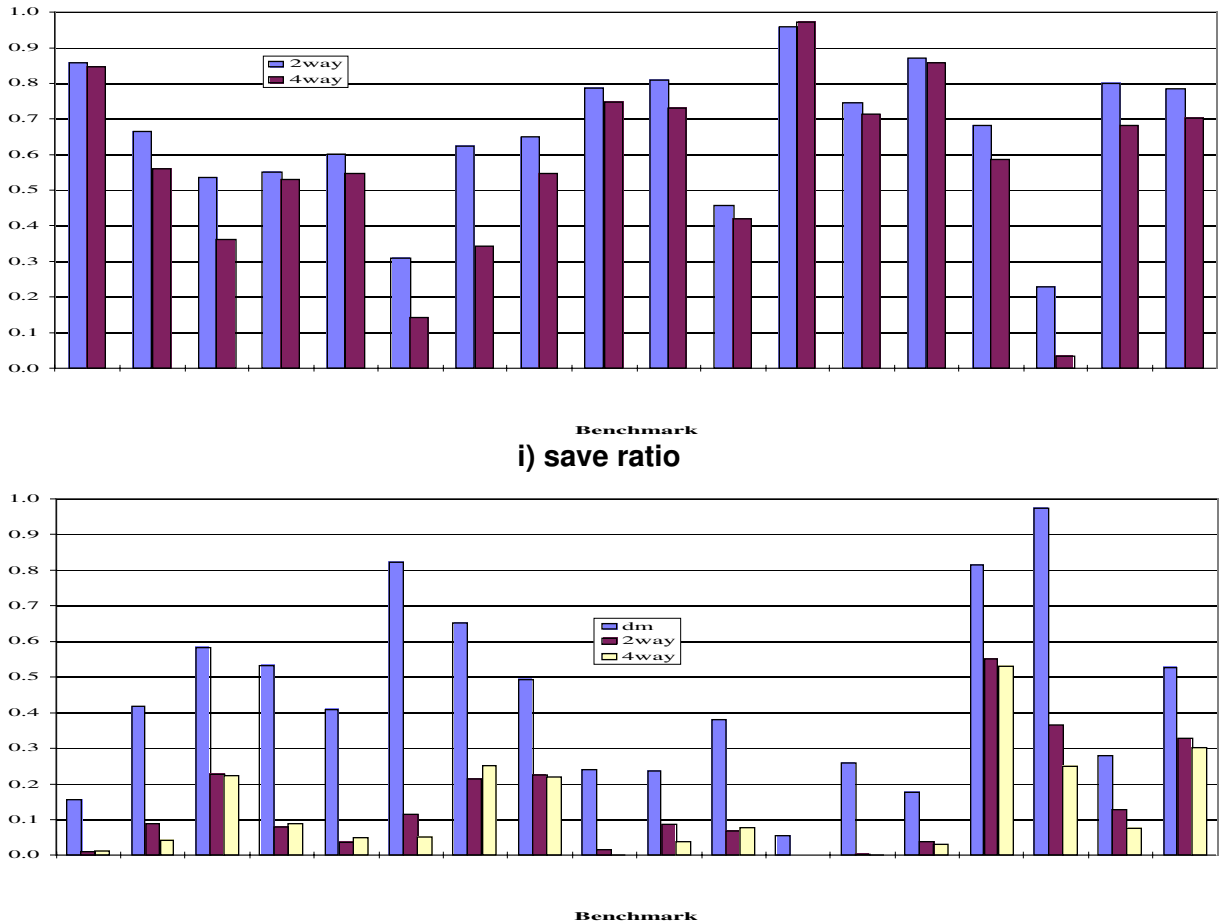


Figure 5: Victim cache save and swap characteristics as A cache associativity increases. The top figure shows the fraction of saves performed as A cache associativity increases relative to the number of saves performed for a direct-mapped A cache Victim configuration. The bottom figure shows the fraction of those saves performed that are swapped back to A due to reaccess.

better usage of the B cache is due to its ability to allow elements to reside in B longer, as it is possible to evict items directly from A; in the Victim scheme, all evicted items from A must first travel through the B cache before they can leave the L1 cache structure. In the SPECfp95 benchmarks, however, this additional time in L1 for each evicted A cache element benefits Victim cache performance, which equals or beats ABC's performance in all the benchmarks except **swim**. Nevertheless, the difference in performance between Victim and ABC over the SPECfp benchmarks is less than 0.5%, and neither scheme has a speedup of more than 4% over the base cache. Note that these performance comparisons are for a Victim cache with 0 cycle swap latency; a realistic implementation of the Victim cache would incur at least a 1 or 2 cycle latency to exchange blocks between the caches for each swap. We have seen that as the swap latency increases from 0 to 2 to 4 cycles, the Victim cache's performance degrades. Since no data movement at all is performed between the caches of the ABC scheme, swap latency is not an issue for ABC. Furthermore, ABC makes its performance gains without requiring the costly inter-cache data path needed by Victim.

As the associativity of the A cache increases, however, the Victim scheme's speedup is equaled or exceeded by the other multilateral schemes, as found in [4]. Victim's drop in relative

performance occurs because the B cache in the Victim scheme is not as well utilized, as the sets in the A cache have less frequent conflicts and tend to generate fewer victims that benefit from being saved in the B cache. This phenomenon is well illustrated in Figure 5. The top of Figure 5 shows the decrease in the number of saves to the B cache by the Victim scheme as the associativity of A increases. This decrease occurs for two reasons: 1) as the associativity of A increases, the A cache typically is able to reduce the number of misses incurred, and thus fewer elements are replaced from A; 2) elements that are hit in B and swapped back to A have a higher likelihood of staying in A since the increased associativity of A can allow them to remain in the same set of A as 1 or 3 other useful blocks. Thus, even those blocks that are worth saving don't need to be saved as often.

The bottom portion of Figure 5 shows the fraction of blocks saved to B that are eventually swapped back to A. This ratio drops significantly as the associativity of A increases, indicating that fewer and fewer elements residing in B are ever actually accessed while in L1. Combining the effects of the two figures, we see that the absolute number of swaps being performed, and thus the number of useful elements found in the B cache of a Victim scheme, decreases greatly as A cache associativity increases. As a result, the performance benefit of the B cache in a Victim scheme degrades as A cache associativity increases; less than 31% of the blocks placed in the B cache overall are actually reaccessed from the B cache when the A cache is 4-way associative.

The ABC scheme continues to perform well, however, since the B cache is used to hold data that is currently being referenced and found to conflict with useful data resident in the A cache; when the data in the conflicting A set is not found to be useful (i.e. it has not been accessed since the last CNR to that A-set), it is evicted from A and returns to the next level of memory; the B cache elements are thus allowed to remain and potentially be reaccessed before their eviction.

The performance of ABC pulls away from Victim in the SPECint benchmarks as the associativity of the A cache is increased, and ABC's performance exceeds Victim's performance in the SPECfp benchmarks. Note that the associative A cache results presented are for a Victim cache with 0 cycle swap latency; again, if a realistic swap latency is incurred, Victim's performance will degrade, as seen in the direct-mapped A cache performance numbers (see Figure 2).

7 Conclusions

In this paper we have introduced a new cache management scheme, Allocation By Conflict (ABC), and compared its performance to the Victim cache, two previously proposed reuse-based multilateral schemes, NTS and PCS, and a multilateral performance baseline, Random. As shown in earlier work, the performance of each of the multilateral schemes is better than associative single structure caches of comparable size and comparable, or even better than traditional caches with the same A cache associativity of nearly twice the size. We have shown that the ABC scheme performs best over all of the reuse-based allocation schemes because it makes its allocation decisions based on the current usage of blocks in cache vs. the past tour usage of incoming miss blocks, as done in NTS and PCS. Furthermore, the ABC scheme outperforms the Victim scheme over the SPECint95 suite, and performs comparably over the SPECfp95 benchmarks when the associativity of the A cache is low. The Random scheme also performs quite well for these cache configurations and sizes; however, its performance is less predictable and is likely to change as the cache configurations are varied. As the associativity of the A cache increases, the Random and Victim schemes' performance degrades relative to NTS, PCS, and ABC.

The ABC scheme has the lowest hardware requirements of any of the evaluated cache schemes, requiring only a single additional bit per block in the A cache. The NTS and PCS

schemes require additional temporality detection bits per block and a Detection Unit for storing reuse information with an associative lookup for making allocation decisions. The Victim scheme requires a costly data path between the caches for swaps and saves, and Random requires a uniform random number generator. NTS and PCS do perform comparably to ABC as the associativity of the A cache increases. However, their implementation complexity is much higher than ABC's, and ABC is the best performing cache scheme overall.

8 References

[1] N.P. Jouppi, "Improving Direct-Mapped Cache Performance by the Addition of a Small Fully Associative Cache and Prefetch Buffers," *Proceedings of ISCA-17*, pp. 364–373, June 1990.

[2] J. A. Rivers, E. S. Tam, and E. S. Davidson, "On Effective Data Supply for Multi-Issue Processors," *Proceedings of the 1997 IEEE International Conference on Computer Design*, October 1997.

[3] J. A. Rivers and E. S. Davidson, "Reducing Conflicts in Direct-Mapped Caches with a Temporality-Based Design," *Proceedings of the 1996 ICPP*, vol. I., pp. 151 - 160, August 1996.

[4] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens, "Utilizing Reuse Information in Data Cache Management," *Proceedings of the 1998 ICS,* pp. 449 - 456, July 1998.

[5] A. Gonzalez, C. Aliagas, and M. Valero, "A Data Cache with Multiple Caching Strategies Tuned to Different Types of Locality," *Proceedings of the 1995 Conference on Supercomputing*, pp. 338–347, July 1995.

[6] T. Johnson and W.W. Hwu, "Run-time Adaptive Cache Hierarchy Management via Reference Analysis," *Proceedings of ISCA-24*, pp. 315–326, June 1997.

[7] D. Stiliadis and A. Varma, "Selective Victim Caching: A Method to Improve the Performance of Direct-Mapped Caches," *IEEE Transactions on Computers*, vol.46, no.5, pp. 603–610, May 1997.

[8] J. E. Bennett and M. J. Flynn, "Prediction Caches for Superscalar Processors," *Proceedings of ISCA-30*, pp. 81–90, December 1997.

[9] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun, "A Modified Approach to Data Cache Management," *Proceedings of MICRO-28*, pp. 93–103, December 1995.

[10] E. S. Tam, "Improving Cache Performance Via Active Management," *Ph.D. Dissertation*, University of Michigan, Ann Arbor, 1999.

[11] D. Burger and T. M. Austin, "Evaluating Future Microprocessors: the SimpleScalar Tool Set," *Technical Report #1342*, University of Wisconsin, June 1997.

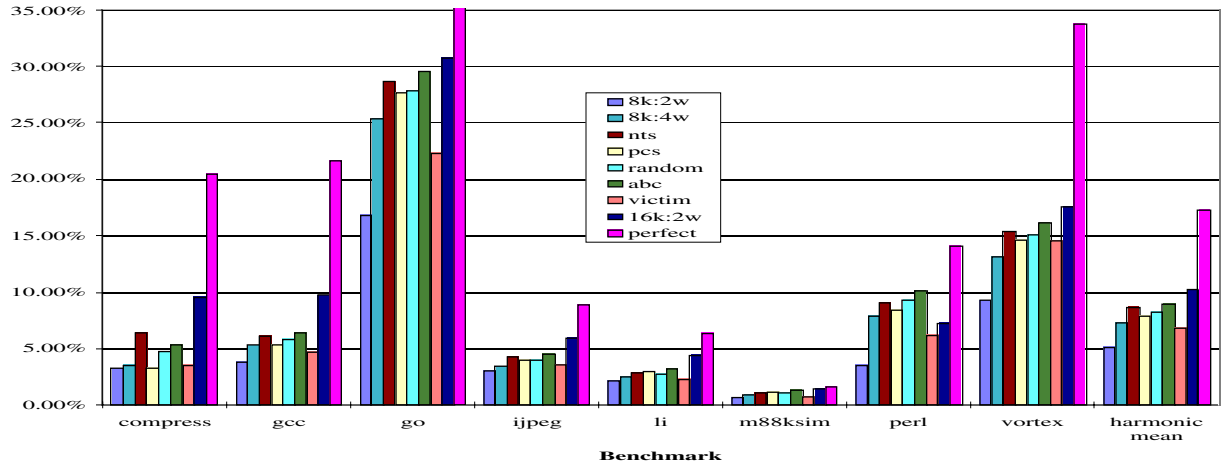
[12] E. S. Tam. J. A. Rivers, G. S. Tyson, and E. S Davidson, "*mlcache:* A Flexible multilateral Cache Simulator," *Proceedings of MASCOTS'98*, pp. 19–26, July 1998.

[13] E. S. Tam, J. A. Rivers, V. Srinivasan, G. S. Tyson, and E. S. Davidson, "Evaluating the Performance of Active Cache Management Schemes," *Proceedings of the 1998 ICCD*, pp. 368–375, October 1998.

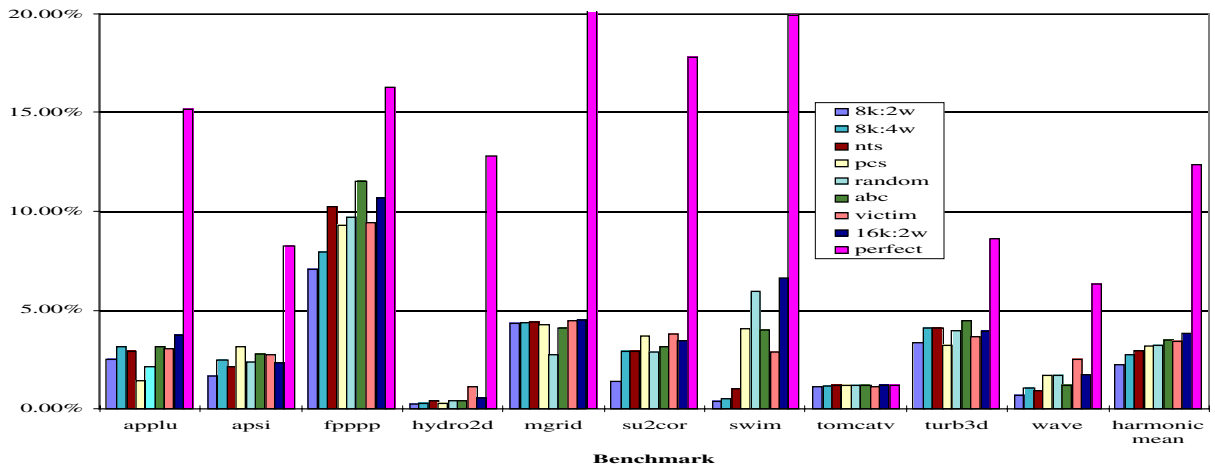
[14] C-C. Lee, I-C. K. Chen, and T. N. Mudge, "The Bi-Mode Branch Predictor," *Proceedings of MICRO-30*, pp. 4–13, December 1997.

9 Appendix

Included below are the detailed speedup graphs for the 2-way and 4-way cache evaluations.

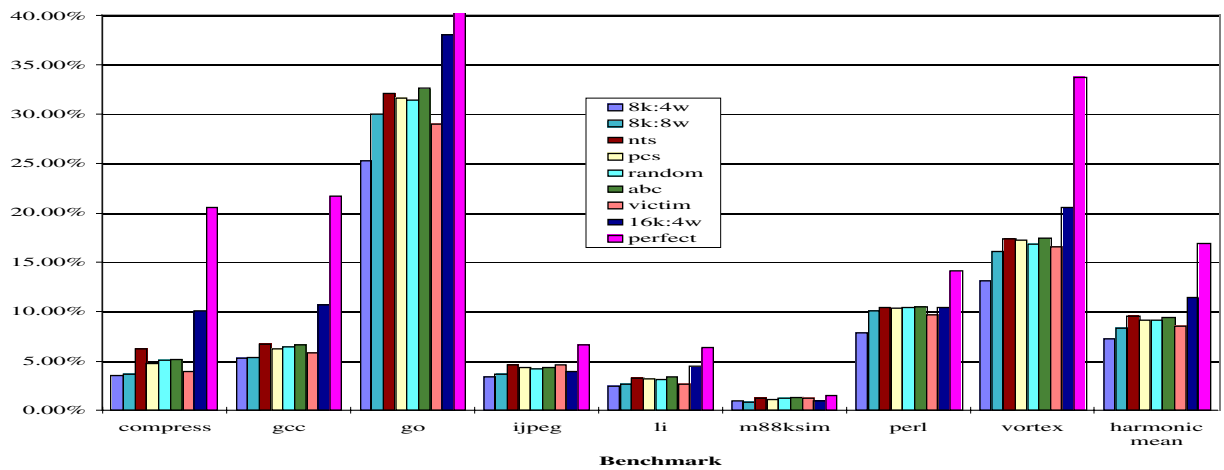


i) SPECint95

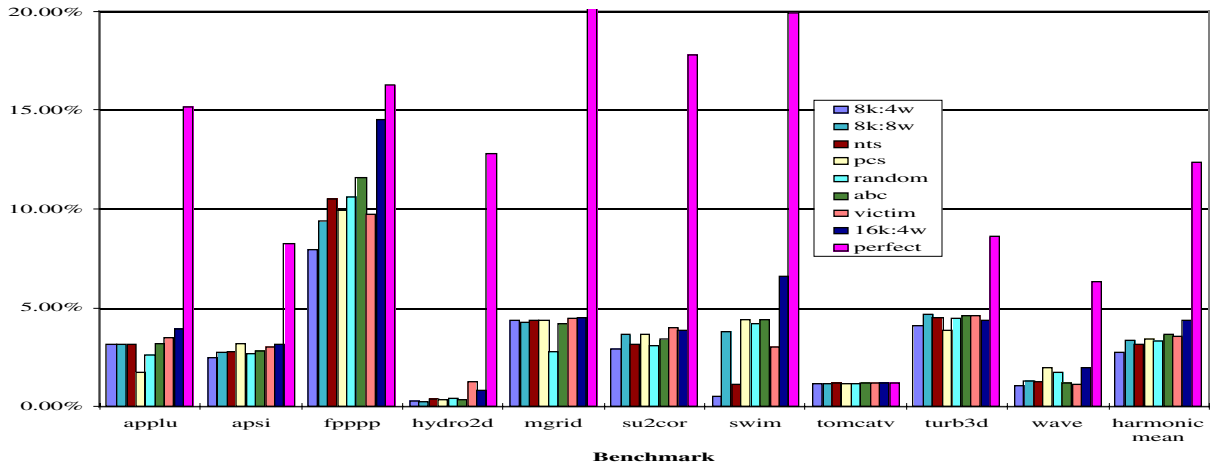


ii) SPECfp95

Figure 6: Overall execution speedup for the six evaluated cache schemes, relative to a single direct-mapped 8K cache (8k:1w). SPECint95 performance is shown on top, SPECfp95 below. Each of the multilateral schemes has a 2-way associative A cache. Perfect is the performance of a perfect cache. The harmonic mean speedup over the SPECint and SPECfp benchmarks is also shown. Victim has 0 cycle swap latency.



i) SPECint95



ii) SPECfp95

Figure 7: Overall execution speedup for the six evaluated cache schemes, relative to a single direct-mapped 8K cache (8k:1w). SPECint95 performance is shown on top, SPECfp95 below. Each of the multilateral schemes has a 4-way associative A cache. Perfect is the performance of a perfect cache. The harmonic mean speedup over the SPECint and SPECfp benchmarks is also shown. Victim has 0 cycle swap latency.