Active Management of Data Caches by Exploiting Reuse Information

Edward S. Tam†, Jude A. Rivers‡[*], Vijayalakshmi Srinivasan†, Gary S. Tyson†, and Edward S. Davidson†

†Advanced Computer Architecture Laboratory, EECS Department
The University of Michigan
Ann Arbor, MI 48109
{estam,sviji,tyson, davidson}@eecs.umich.edu

‡T. J. Watson Research Center
IBM Corporation
P. O. Box 218
Yorktown Heights, NY 10598
rivers@watson.ibm.com

Abstract

As microprocessor speeds continue to outpace memory subsystems in speed, minimizing average data access time grows in importance. Multi-lateral caches afford an opportunity to reduce the average data access time by active management of block allocation and replacement decisions. We evaluate and compare the performance of traditional caches and multi-lateral caches with three active block allocation schemes: MAT, NTS, and PCS.

We also compare the performance of NTS and PCS to multi-lateral caches with a near-optimal, but nonimplementable policy, *pseudo-opt*, that employs future knowledge to achieve both active allocation and active replacement. NTS and PCS are evaluated relative to *pseudo-opt* with respect to miss ratio, accuracy of predicting reference locality, actual usage accuracy, and tour lengths of blocks in the cache. Results show the multi-lateral schemes do outperform traditional cache management schemes, but fall short of *pseudo-opt*; increasing their prediction accuracy and incorporating active replacement decisions would allow them to more closely approach *pseudo-opt* performance.

Keywords: *multi-lateral cache, active management, reuse information*

1 Introduction

Minimizing the *average data access time* is of paramount importance when designing high-performance machines. Unfortunately, access time to off-chip memory (measured in processor clock cycles) has increased dramatically as the disparity between main memory access

[*]Work done while at the University of Michigan.

times and processor clock speeds widen. The effect of this disparity is further compounded as multiple-issue processors continue to increase the number of instructions that can be issued each cycle. There are many approaches to minimizing the average data access time. The most common solution is to incorporate multiple levels of cache memory on-chip, but still allocate and replace their blocks in a manner that is essentially the same as when caches first appeared three decades ago.

Recent studies [23][16][10][8][18] have explored better ways to configure and manage a resource as precious as the first-level (L1) cache. Active cache management (active block allocation and replacement) can improve the performance of a given size cache by maintaining more useful blocks in the cache; active management retains reuse information from previous tours of blocks and uses it to manage block allocations and/or replacements in subsequent tours[1]. In order to partition the allocation of blocks within a cache structure, several proposed schemes [16][10][8][18] incorporate an additional data store within the L1 cache structure and intelligently manage the state of the resulting *multi-lateral*[2] [17] cache by exploiting reuse pattern information. These structures perform active block allocation, but still relegate block replacement decisions to simple hardware replacement algorithms. While processor designers typically design for the largest possible caches that can still fit on the ever growing processor die, multi-lateral designs have been shown to perform as well as or better than larger, single structure caches while requiring less die area [17][22]. For a given die size, reducing the die requirements to attain a given rate of data supply can free that space for other resources – for example, to dedicate more space to branch prediction, data forwarding, instruction supply, and the instruction reorder buffer.

In this paper we evaluate the performance of three proposed cache schemes that perform active block allocation and compare their performance to one another and to traditional single-structure caches. We implement the MAT [10], NTS [16], and PCS [18] schemes using

[1]A tour of a cache block is the time interval between an allocation of the block in cache and its subsequent eviction. A given memory block can have many tours through the cache.

[2]We use the term *multi-lateral* to refer to a level of cache that contains two or more data stores that have disjoint contents and operate in parallel.

hardware that is as similar as possible in order to do a fair comparison of the block allocation algorithms that each uses. Our experiments show that making placement decisions based on effective address-based block reuse, as in the NTS scheme, outperforms the macroblock-based and PC-based approaches of MAT and PCS, respectively. All three schemes perform comparably to larger direct-mapped caches and better than associative caches of similar size.

We then examine the performance of optimal and near-optimal multi-lateral caches to determine the performance potential of multi-lateral schemes. Optimal and near-optimal schemes excel in block replacement decisions, while their block allocation decisions are a direct consequence of the replacement decision. We compare the performance of two implemented multi-lateral schemes to the near-optimal scheme to determine the reason for their performance. For the implemented schemes to perform better, improvements need to be made in their block allocation and replacement choices.

The rest of this paper is organized as follows. Section 2 discusses techniques that aid in reducing the average data access time. Section 3 discusses active cache management in detail and presents past efforts to perform active block allocation. Section 4 presents our simulation methodology and Section 5 evaluates the performance of the three multi-lateral schemes. In Section 6, we present the performance of optimal and near-optimal multi-lateral schemes, which perform (near-)optimal replacement of blocks, and compare the decisions made in the near-optimal scheme to those made in two of the implementable schemes. Conclusions are given in Section 7.

2 Background

There are many techniques for reducing or tolerating the average memory access time. Prominent among these are: 1) store buffers, used to delay writes until bus idle cycles in order to reduce bus contention; 2) non-blocking caches, which overlap multiple load misses while fulfilling other requests that hit in the cache [19][12]; 3) hardware and software prefetching methodologies that attempt to preload data from memory to the cache before it is needed [5][1][4][6][15]; and 4) victim caching [11], which improves the performance of direct mapped caches through the addition of a small, fully associative cache between the L1 cache and the next level in the hierarchy. While these schemes do contribute to reducing average data access time, this paper approaches the problem from the premise that the average data access time can be reduced by exploiting reuse pattern information to actively manage the state of the L1 cache. This approach can be used with these other techniques to reduce the average data access time further.

3 Active Cache Management

Active cache management can be used to improve the performance of a given size cache structure by controlling the data placement and management in the cache to keep the active working set resident, even in the presence of transient references. Active management of caches consists of two parts: allocation of blocks within the cache structure on a demand miss[3] and replacement of blocks currently resident in the cache structure[4]. Block allocation in today's caches is passive and straightforward: blocks that are demand fetched are placed into their corresponding set within the cache structure. However, this decision does not take into consideration the block's usefulness or usage characteristics. Examining the past history of a given block is one method of aiding in the future allocations of the block. Decisions can range from simply not caching the target block (bypassing) to placing the block in a particular portion of the cache structure, with the hope of making best use of the target cache block and the blocks that remain in the cache.

Simple block replacement policies are used to choose a block for eviction in today's caches, and this choice is often suboptimal. For a multi-lateral cache, the allocation and replacement problems are coupled. In particular, the blocks that are available for replacement are a direct consequence of the allocation of a demand-missed block.

Recently, several approaches to more efficient management of the L1 data cache via block

[3]Allocation decisions for blocks not loaded on a demand miss, e.g. prefetched blocks in a streaming buffer scheme as proposed in [11], and bypassing schemes are not considered here. However, schemes that make proper bypass decisions and allocation decisions for prefetched data can further improve upon the performance of the schemes evaluated herein.

[4]We consider only write-allocate caches in this paper. Write no-allocate caches follow a subset of these rules, where writes are not subject to these allocation decisions.

allocation decisions have emerged in the literature: NTS [16], MAT [10], Dual/Selective [8], and PCS [18]. However, none of these approaches makes sophisticated block replacement decisions, and instead relegates these decisions to their respective cache substructures.

3.1 The NTS Model

The NTS (nontemporal streaming) cache [16] 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 a separate small cache placed in parallel with the main L1 cache; all other blocks (those marked T and those for which no prior information is available) are handled in the "main" 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.

3.2 The MAT Model

The MAT (memory address table) cache [10] is another scheme based on the use of effective addresses; however, it dynamically partitions cache data blocks into two groups based on their frequency of reuse. Blocks become tagged as either Frequently or Infrequently Accessed. A memory address table is used to keep track of reuse information. The granularity for grouping is a *macroblock*, defined as a contiguous group of memory blocks considered to have the same usage pattern characteristics. Blocks that are determined to be Infrequently Accessed are allocated in a separate small cache. This scheme has shown significant speedups over generic caches due to improved miss ratios, reduced bus traffic, and a resulting reduction in the average data access latency.

3.3 The Dual Cache/Selective Cache Model

The Dual Cache [8] has two independent cache structures, a spatial cache and a temporal cache. Cache blocks are dynamically tagged as either temporal or spatial. A locality prediction table is used to maintain information about the most recently executed load/store instruction. The blocks that are tagged neither spatial nor temporal do not find a place in the cache and bypass the cache. This method is more useful in handling vector operations which have random access patterns or very large strides and introduce self interference. However, its two caches do not necessarily maintain disjoint contents. The temporal cache is designed to have a smaller line size compared to the spatial cache. If the required data is found in both caches it is read from the temporal cache or written into both in parallel. In order to overcome this replication and coherence problem, the authors proposed a simplified version of the Dual Cache, called the Selective Cache. The Selective Cache has only one memory unit like a conventional cache, but incurs more hardware cost due to its locality prediction table, as in the Dual Cache. Only data exhibiting spatial locality or temporal locality that is not self-interfering is cached. For most of the benchmarks in their study, this scheme was shown to perform better than a conventional cache of the same size. The Selective Cache itself is an improvement of the Bypass Cache [7], which relies on compiler hints to decide whether a block is to be cached or bypassed.

3.4 The PCS Model

The PCS (program counter selective) cache [18] is a multi-lateral cache design that evolved from the CNA cache scheme [23]. 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 brought to the cache, 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.

3.5 Other Multi-Lateral Cache Schemes

Several other cache schemes can be considered multi-lateral caches, such as the Assist cache [13] used in the HP PA-7200, and the Victim cache [11]. However, neither of these schemes actively manage their cache structures using reuse information obtained dynamically during program execution. The Assist cache uses a small data store as a staging area for data entering the L1 and potentially prevents data from entering the L1 when indicated by a compiler hint. The Victim cache excels in performance when a majority of the cache misses are conflict misses which result from the limited associativity of the main cache; the buffer in the Victim scheme serves to dynamically increase the associativity of a few hot spots in the (typically direct-mapped) main cache. While both schemes have been shown to perform well [22], they each require a costly data path between the two data stores to perform the data migrations they require. Without the inter-cache data path present, these schemes cannot operate, as they use no previous tour information for actively deciding which data store to allocate a block to. While the Victim cache has been shown to perform well relative to the above actively-managed schemes when the main cache is direct-mapped [18], in this paper we evaluate only multi-lateral schemes that use dynamic information to allocate data among two data stores with no direct data path between them.

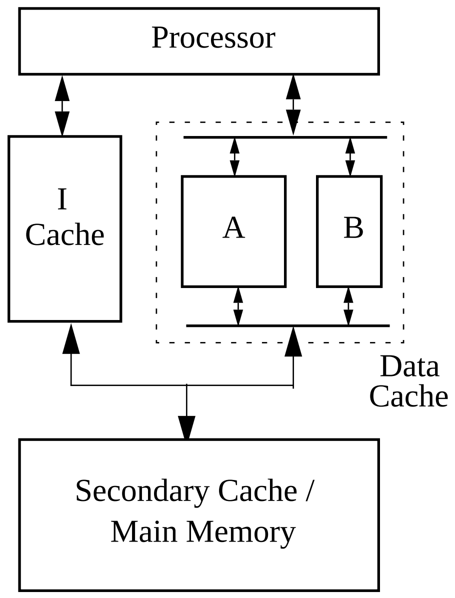
4 Simulation Methodology

A simulator and a set of benchmark programs were used to compare the performance of the multi-lateral cache strategies. This section describes the dynamic superscalar processor and memory simulators used to evaluate these cache memory structures, the system configuration used, and the methods, metrics, and benchmarks that constitute the simulation environment.

4.1 Processor and Memory Subsystem

The processor modeled in this study is a modification of the *sim-outorder* simulator in the *SimpleScalar* [3] 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* tool discussed below, 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. The data cache is modeled as



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
F. U. 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	Multi-Lateral 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.

a conventional data cache split into two subcaches (A and B with disjoint contents) and placed in parallel within L1. In this multi-lateral cache, each subcache is unique with its own configuration: size, set-associativity, replacement policy, etc. The A and B caches are probed in parallel, and are equidistant from the CPU. Both A and B are non-blocking with 32-byte lines and single cycle access times. A standard (single-structured) 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 management strategies for the L1.

4.2 The *mlcache* Simulation Tool

mlcache [22] is an event-driven, timing-sensitive cache simulator based on the Latency Effects (LE) cache timing model, discussed in depth in [21]. It can be easily configured to model various single and multi-lateral cache structures by using its library of cache state and data movement routines. For interactions not modeled in the library routines, users can write

Support Routine	**Description**
check_for_cache_hit()	check to see if an accessed block is present in the cache
update()	place an accessed block into the cache
move_over()	move an accessed block from one cache to another
do_swap()	move an accessed block from cache1 to cache2 and move the evicted block to cache1
do_swap_with_inclusion()	place an accessed block into both cache1 and cache2 and move the evicted block from cache2 to cache1
do_save_evicted()	move the block evicted from cache1 to cache2
find_and_remove()	remove a block from a cache
check_for_reuse()	determine if a block exhibits temporal behavior (word reuse)

Table 1: The basic support routines provided with the *mlcache* simulator. The user can call these routines from a configuration file to control the cache state and interactions.

their own management routines and call them from the simulator. The tool can be easily joined to a wide range of event-driven processor simulators. As described above, our processor model in this work is based on the *SimpleScalar* toolset. Together, a combined processor-and-cache simulator, such as *SimpleScalar+mlcache*, can provide detailed evaluations of multiple cache designs running target workloads on proposed processor/cache configurations.

mlcache is easily retargetable due to the provision of a library of routines that a user can choose from to perform the actions that should take place in the cache in each situation. The routines are accessed from a single C file, named *config.c*. The user simply modifies *config.c* to describe all of the desired interactions between the caches, processor, and memory. The user also controls when the actions occur via the delayed update mechanism built into the cache simulator. Delayed update is used to allow a behavioral cache simulator, such as *DineroIII* [9], to account for latency for latency- or latency-adding effects. The use of delayed update causes the effects of an access, i.e. an access' placement into the cache, the removal of the replaced block, etc. to occur only after the calculated latency of the access has passed. Table 1 shows the routines provided and a brief description of each. If more interactions are needed than these, additional library routines can be added. However, from these brief examples it is easy to see that this modular, library-based simulator already allows a significant range of cache configurations to be examined.

We evaluate the performance of three of the multi-lateral schemes, MAT, NTS, and PCS, and compare their performance to three traditional, single-structure caches: a 16K direct-mapped cache, a 16K 2-way associative cache, and a 32K direct-mapped cache. The configurations of the evaluated caches are shown in Table 2.

4.3 Simulated Cache Schemes

Performing a realistic comparison among the program counter and effective address schemes requires detailed memory simulators for the MAT, NTS, and PCS cache management schemes described above. We chose to omit the Selective Cache, as its block allocation decisions are similar to those made by PCS, while PCS' hardware implementation is simpler. To ensure a fair comparison and evaluation, we placed all the management schemes on the same platform within a uniform multi-lateral environment, using the *mlcache* tool. Each of the configurations includes a 32-entry structure that stores reuse information, as described for each scheme.

The following subsections describe our implementations of the MAT, NTS, and PCS cache management schemes. The main cache is labeled cache A, the auxiliary buffer is labeled cache B, and both caches are placed equidistant from the CPU. The three schemes are configured to be as similar as possible to one another so that their performance differences can be attributed primarily to differences among the block allocation decisions that they make.

4.3.1 Structure and Operation of the NTS Cache

The NTS cache, using the model in [18], which was adapted from the scheme proposed in [16], actively allocates data within L1 based on each block's usage characteristics. In particular, blocks known to have exhibited only nontemporal reuse are placed in B, while the others (presumably temporal blocks) are sent to A. This is done in the hope of allowing temporal data to remain in the larger A cache for longer periods of time, while shorter lifetime nontemporal data can for a short while be quickly accessed from the small, but more associative B cache.

On a memory access, if the desired data is found in either A or B, the data is returned to the processor with 0 added latency, and the block remains in the cache in which it is

	Single	MAT		NTS		PCS	
Cache	A	A	B	A	B	A	B
Size	16K/16K/32K	16K	2K	16K	2K	16K	2K
Associativity	1/2/1	1	full	1	full	1	full
Replacement policy	–/LRU/–	–	LRU	–	LRU	–	LRU
latency to next level	18	18	18	18	18	18	18

Table 2: Characteristics of the four configurations studied. Times/latencies are in cycles.

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 and is managed as follows. 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. If no corresponding DU entry is found for the evicted block, a new DU entry is created and made MRU in the DU structure. On a miss, if the new (missed) 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 temporal, and B if not. The DU entry is then made MRU in the DU so that it has a better chance of remaining in the DU for future allocation predictions. Thus, each creation or access of an entry in the DU is treated as a "use" and the DU (with 32 entries, in these simulations) is maintained with LRU replacement. If no matching DU entry is found, the missed block is assumed to be temporal and placed in A.

4.3.2 Structure and Operation of the PCS cache

The PCS cache [18] decides on data placement 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, the performance of blocks missed by individual memory accessing instructions, rather than individual data blocks, determines the placement of data in the PCS scheme.

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 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. If no DU entry matches that PC value, one is created and replaces the LRU entry in the DU. If that instruction subsequently misses, the loaded block is placed in B if the instruction's PC hits in the DU and the prediction bit indicates NT; otherwise the block is placed in A. If the instruction misses in the DU, the data is placed in A.

4.3.3 Structure and Operation of the MAT cache

The MAT cache [10] structure has a Memory Address Table (MAT) for keeping track of reuse information and for guiding data block placement into the A or B cache of the L1 structure. In this implementation, the MAT is a 32-entry fully associative structure (like the DU in NTS and PCS). Note, however, that the original implementation of MAT (reported in [10]) used a 1K entry direct mapped table. Each MAT entry consists of a macroblock address and an n-bit saturating counter. An 8-bit counter and a 1KB macroblock size is used here, as in the original study.

On a memory access, caches A and B are checked in parallel for the requested data. At the same time, the counter in the corresponding MAT entry for the accessed block is incremented; if there is no corresponding entry, one is created, its counter is set to 0, and the LRU entry in the MAT is replaced. This counter serves as an indicator of the "usefulness" of a given macroblock, and is used to decide whether a block in that macroblock should be placed in the A or B cache during its next tour.

On a cache miss, the macroblock address of the incoming block is used as an index into the MAT. If an entry exists, its counter value is incremented and compared against the decremented counter of the macroblock corresponding to the block that would be replaced if the incoming block were to be placed in the A cache. The counter is decremented to ensure that that data can eventually be replaced; the counter of the resident data will continue to decrease if it is not reaccessed often enough and it continues to conflict with more recently accessed blocks. If the counter value of the incoming block is higher than that of the conflicting block currently in cache A, the incoming block replaces this block in the A cache. This situation indicates that the incoming block is in a macroblock that has shown more "usefulness" in earlier tours than the macroblock in which the conflicting block resides, and should thus be given a higher priority for residing in the larger main cache. If the counter value of the incoming block is less than that of the current resident block, the incoming block is placed in the smaller B cache.

Finally, if no entry corresponds to the incoming block, the block is placed in the A cache by default and a new entry is created for it in the MAT, with its counter initialized to zero. If no entry corresponds to the conflicting block currently in cache A, its counter value is assumed to be 0, permitting the new block to replace it easily. When no entry is found in the MAT for a resident block in cache A, another macroblock that maps to the same set in the MAT must have been accessed more recently, and the current block is therefore less likely to be used in the near future.

As with the NTS and PCS schemes, there is no direct data path between the A and B caches. Unlike those schemes, however, the MAT structure is updated for every access to the cache instead of only on replacements.

4.4 Benchmarks

Table 3 shows the 5 integer and 3 floating point programs from the SPEC95 benchmark suite used in this study. These programs have varying memory requirements, and the simulations were done using the training data sets. Each program was run to completion (with the exception of **perl**, which was terminated after the first 1.5 billion instructions).

4.5 The Relative Cache Effects Ratio

An important metric for evaluating any cache management scheme is the cache hit/miss ratio. However, in OOO processors with multi-ported non-blocking caches, effective memory latencies (as seen by the processor) vary according to the number of outstanding miss requests. Since the main focus of this study is to evaluate the effectiveness of the L1 cache

Program	Instruction Count (millions)	Memory References (millions)		Perfect Memory Performance	
		Loads	Stores	Cycle Count (millions)	IPC
SPEC95 Integer Benchmarks					
Compress	35.68	7.37	5.99	5.35	6.6644
Gcc	263.85	61.15	36.24	43.50	6.0648
Go	548.13	115.79	41.40	91.33	6.0049
Li	956.49	286.38	168.79	151.32	6.3210
Perl	1,500.00	396.82	269.83	232.89	6.4408
SPEC95 Floating Point Benchmarks					
Hydro2d	974.50	196.11	60.90	127.63	7.6353
Su2cor	1,054.09	262.20	84.74	152.34	6.9192
Swim	849.92	205.18	58.44	113.02	7.5201

Table 3: The eight benchmarks and their memory characteristics.

structure using special management techniques, the Relative Cache Effects Ratio (RCR) was developed [17]. The RCR for a given processor running cache configuration X relative to cache configuration base, is given by:

$$RCR_X = \frac{CycleCount_X - CycleCount_{PerfectCache}}{CycleCount_{base} - CycleCount_{PerfectCache}} \quad (1)$$

where CycleCount$_{PerfectCache}$ is the total number of cycles needed to execute the same program on the same processor with a perfect cache configuration. RCR, a normalized metric between the base cache and the perfect cache, is 1 for the base cache configuration and 0 for the perfect cache configuration. Cache configurations that perform better than the base have RCR between 0 and 1, with lower RCR being better. A cache configuration that performs worse than the base has RCR > 1. RCR gives an indication of the finite cache penalty reduction obtained when using a given cache configuration. RCR mirrors the performance indicated by speedup numbers, but isolates the cache penalty cycles from total run time and rescales them as a fraction of the penalty of a traditional (base) cache. It thus gives a direct indication of how well the memory subsystem performs relative to an ideal (perfect) cache. In addition to overall performance speedup, this metric will be used to measure the relative performance gains of each cache management approach in Section 5.

4.6 The Block Tour and Reuse Concept

The effectiveness of a cache management scheme can also be measured by its ability to minimize the cumulative number of block tours during a program run. Individual cache block tours are monitored and classified based on the reuse patterns they exhibit. A tour that sees a reuse of any word of the block is considered dynamic temporal; a tour that sees no word reuse is dynamic nontemporal. Both dynamic temporal and dynamic nontemporal tours can be further classified as either spatial (when more than one word was used) or nonspatial (when no more than one word was used). This allows us to classify each tour into one of four data reuse groups: 1) nontemporal nonspatial (NTNS), 2) nontemporal spatial (NTS), 3) temporal nonspatial (TNS), and 4) temporal spatial (TS). Good management schemes should result in fewer (longer) tours, and a consequently higher percentage of data references to blocks making TS tours. NTNS and NTS tours are problematic; more frequent references to such data are likely to cause more cache pollution. To minimize the impact of bad tours, a good multi-lateral cache management scheme should utilize an accurate block behavior prediction mechanism for data allocation decisions.

5 Experimental Results

Miss ratio is often used to rank the performance benefits of particular cache schemes. However, miss ratio is only weakly correlated with the performance of latency-masking processors with non-blocking caches. Furthermore, it fails to capture the latency-adding effect of delayed hits on overall performance. Delayed hits, discussed in [22][21], are accesses to data that are currently returning to the cache on behalf of an earlier miss to that cache block. Delayed hits incur latencies larger than cache hits, but generally less than a full cache miss, as the requested data is already in transit from the next level of memory. Two programs exhibiting similar miss ratios may thus have quite different overall execution times due to differing numbers of delayed hits and the extent of latency masking, as shown in [22].

To avoid oversimplifying a cache scheme's impact on overall performance, we instead concentrate on two metrics from timing-sensitive experiments: overall speedup relative to a

	Compress	Gcc	Go	Hydro2d	Li	Perl	Su2cor	Swim
MAT	0.220	0.074	0.031	0.479	0.041	0.060	0.256	0.220
NTS	0.219	0.070	0.021	0.474	0.043	0.032	0.257	0.230
PCS	0.221	0.086	0.024	0.494	0.050	0.033	0.257	0.167
16K:1w	0.239	0.112	0.070	0.484	0.062	0.062	0.269	0.228
16K:2w	0.217	0.061	0.038	0.488	0.038	0.044	0.248	0.224
32K:1w	0.215	0.071	0.035	0.452	0.029	0.051	0.244	0.161

Table 4: Miss ratios of the 6 cache schemes running the 8 benchmarks.

base cache and the Relative Cache Effects Ratio, presented above.

5.1 Miss Ratio

Table 4 shows the miss ratios for each of the six cache configurations when running the eight benchmarks, and Figure 2 shows the corresponding speedup relative to a 16K direct-mapped cache. Naively, we might assume that when comparing two configurations for a particular application, a higher miss ratio would imply a lower speedup, and that (since cache stalls account for only a portion of the run time) the relative speedup would be less than the relative miss ratio. However, a comparison of Table 4 and Figure 2 shows that this assumption is not valid. In **compress**, for example, the miss ratio of PCS is about 1.01x that of NTS, but its run time is about 1.04x longer. In **gcc**, the 32K direct-mapped cache actually has a higher miss ratio, but less run time than the 16K 2-way associative cache. In **swim**, NTS has a higher miss ratio, but less run time than the MAT, 16K and 32K direct-mapped, and 16K 2-way caches. Thus, relative miss ratio alone is an inadequate indicator of relative performance; latency masking, miss latency overlap, and delayed hits must be incorporated in a timing model to get an accurate performance assessment. Therefore, we concentrate our performance analysis on latency-sensitive metrics, such as speedup and RCR.

5.2 Speedup

The speedup achieved by each scheme for each program is shown in Figure 2, where the single 16K direct-mapped (16k:1w) cache is taken as the base. Overall, the speedup obtained by using the multi-lateral cache schemes ranges from virtually none in **hydro2d** to just over 16% in **go** with NTS. Clearly, some of the benchmarks tested do not benefit from any of the improvements offered by the cache schemes evaluated, i.e. better management of the L1

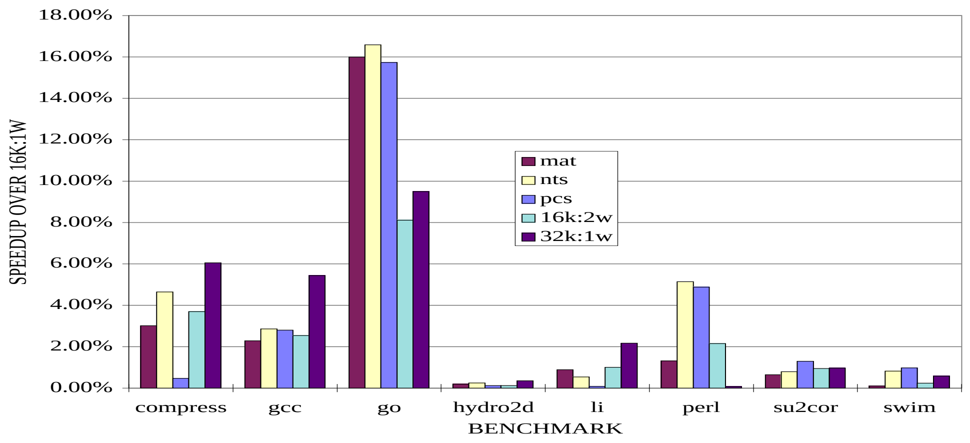


Figure 2: Overall execution time speedup for the five evaluated cache schemes, relative to a single direct-mapped 16K cache (16K:1w).

data store by the multi-lateral schemes, increased associativity of a single cache (16k:2w), or a larger cache (32k:1w). In benchmarks where there is appreciable performance gain over the base cache, the multi-lateral schemes often perform as well as or better than either a higher-associative single cache or a larger direct-mapped cache. In **compress** and **gcc**, the benchmarks' larger working sets benefit from the larger overall cache space provided by the 32K direct-mapped structure, although even for these benchmarks the multi-lateral schemes are able to obtain a significant part of the performance boost via their better management of the cache. Despite their smaller size, the multi-lateral caches generally perform well compared to the larger 32K direct-mapped cache and are generally faster than the 2-way associative cache. In the multi-lateral schemes, the larger direct-mapped A cache offers fast access, and the smaller, more associative B cache can still be accessed quickly due to its small size. Our experiments show that using an 8-way associative B cache instead of a fully associative B cache would reduce performance by less than 1%.

Among the multi-lateral schemes, we see that the NTS scheme provides the greatest speedup in all benchmarks except for **li** (where MAT performs best), **su2cor**, and **swim** (where PCS performs best), but the best multi-lateral cache speedups are only on the order of 1% for these three benchmarks. Both the MAT and PCS schemes can perform well when groups of blocks exhibit similar reuse behavior on consecutive tours through the cache.

The NTS scheme may, however, fail to detect these reuse patterns because it correlates its reuse information to individual cache blocks, as opposed to macroblock memory regions in MAT or to memory accessing instructions in PCS. Thus, the MAT and PCS schemes can perform well if programs exhibit SIMD (single-instruction, multiple-data) behavior, where the reference behavior of nearby memory blocks or blocks referenced by the same memory accessing instruction may be a better indicator of reuse behavior than the usage of an individual block during its last tour. However, the NTS scheme is still competitive in these three benchmarks, and thus gives the best overall performance of these schemes over the full suite of benchmarks.

5.3 RCR Performance

Figure 3 shows the RCR performance of MAT, NTS, PCS, and the two single-structure caches, where the 16K direct-mapped cache serves as the base for comparison. We see here that NTS and PCS eliminate more than 50% of the finite cache penalty experienced by **go** and **perl**. In **compress**, **gcc**, and **li**, the 32K single-structure direct-mapped cache performs best. However, the difference in RCR between it and the best-performing multi-lateral scheme is not very large, except for **li**, where it reduces the finite cache penalty more than twice as much as MAT, the best multi-lateral scheme for this benchmark. None of the caches show a significant improvement in RCR for the remaining three benchmarks (**hydro2d**, **su2cor**, and **swim**).

In some instances, a multi-lateral scheme can experience poor performance, e.g. MAT in the **perl** benchmark, relative to the other multi-lateral schemes. The block allocation scheme of MAT is not well matched to the characteristics of the **perl** benchmark. If many of the blocks are short-lived, but frequently accessed, those blocks will be placed in the smaller, 2K fully-associative cache. If this behavior continues, many blocks will contend for space in the smaller 2K fully associative cache while the larger, 16K cache is badly underutilized. Such a phenomenon can occur in any of these multi-lateral cache schemes; in the extreme, the multi-lateral scheme's performance may degrade to that of the B cache by itself. This

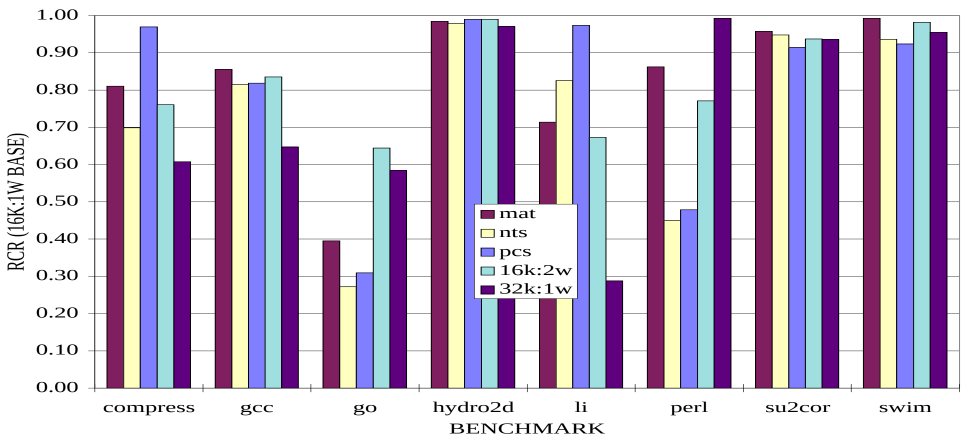


Figure 3: RCR performance of the evaluated configurations running the eight benchmarks. RCRs near 1.0 have performance similar to the base 16K direct-mapped cache while RCRs closer to 0 approach the performance of a perfect cache.

performance degradation can be addressed by improved block allocation mechanisms, as discussed in Section 6.

5.4 Performance Differences and Their Causes

Though MAT and NTS both make block allocation decisions based on the effective address of the block being accessed, their performance differs. MAT may make poor allocation decisions when either the missed block or the block it would replace in cache A has a markedly different desirability than the perceived desirability of the macroblock in which it resides. When such a disparity in desirability occurs, a block-based desirability mechanism, such as that used in NTS, will perform better. Table 5 presents a tour analysis for **go** and **su2cor**. The performance of NTS relative to MAT is well demonstrated in **go**. Not only does NTS appear to manage the tours in this program better, but it actually reduces the number of tours that MAT experiences by 32%. In the case of **su2cor**, where the performance difference between MAT and NTS is small (in terms of RCR), it is clear that the application itself has much more nontemporal spatial data ($> 17\%$), and neither scheme reduces the tours seen by a 16KB direct-mapped cache by more than 6%.

The tour analysis for the two single-structure caches is also shown for these two widely disparate benchmarks. The analyses used to compare MAT and NTS can also be used to compare multi-lateral caches against single-structure caches. In **go**, the 32K direct-mapped

Cache Management Scheme	Total # of Tours	% Reduction in Tours	Total Percentage References to Tour Groups			
			NTNS	NTS	TNS	TS
16K:1w	6,694,690	–	1.27	1.00	11.43	86.29
MAT	2,807,282	58.06	0.48	0.63	5.63	93.26
NTS	1,894,991	71.69	0.21	0.51	4.70	94.58
16K:2w	3,932,345	41.26	0.57	0.61	8.99	89.84
32K:1w	3,450,640	48.46	0.65	0.61	5.54	93.20

Cache Management Scheme	Total # of Tours	% Reduction in Tours	Total Percentage References to Tour Groups			
			NTNS	NTS	TNS	TS
16K:1w	24,848,705	–	0.35	17.98	7.77	73.90
MAT	23,975,325	3.51	0.11	16.96	5.96	76.02
NTS	23,471,060	5.54	0.23	17.36	7.77	74.64
16K:2w	22,824,448	8.15	0.09	17.15	6.95	75.81
32K:1w	23,670,694	4.74	0.26	17.14	6.97	75.63

Table 5: Tour analysis for **Go** (top) and **Su2cor** (bottom).

cache has the best single-structure cache performance; though it has a slightly lower percentage of TS tours than the NTS cache, it has substantially more overall tours, and thus worse overall performance. As with MAT and NTS in **su2cor**, the 2-way associative and 32K direct-mapped caches fail to significantly improve performance over the base cache due to the high percentage of NTS data accessed.

NTS performs better than PCS overall in both speedup and RCR. The different basis for decision making used by PCS and NTS (PC and effective address, respectively), results in different performance. The PCS scheme may place a block suboptimally in the cache since its placement is influenced by other blocks previously referenced by the requesting PC. For example, a set of blocks may be brought into the cache by one instruction at the beginning of a large routine. These blocks may be reused in different ways during different parts of program execution (e.g. temporal during an initialization phase and nontemporal during the main program's execution). All of these blocks' usage characteristics may be attributed to a single entry in the DU, tied to the PC of the instruction that brought the blocks to the cache. When each of these tours end, the instruction's entry in the DU is updated with that particular tour's behavior, directly affecting the placement of the next block requested by this instruction. In effect, the allocation decisions of PCS are influenced by the most recently replaced block that is associated with the load instruction in question; if the characteristics of those most recently replaced blocks is not persistent, as discussed in Section 6.3.4, the allocation decisions made by PCS for this load instruction will vary often, potentially degrading performance.

However, program counter management schemes may be good if a given instruction loads data whose usage is *strongly biased* [14] in one direction, i.e. if these tours are almost all temporal or almost all nontemporal. In this case, accurate behavior predictions for future tours will result in good block placement for that instruction. However, if the data blocks loaded by the instruction have differing usage characteristics (i.e. *weakly biased* [14]) then placement decisions of its blocks will be poor.

Block usage history (T/NT) is kept in a single bit (NTS and PCS); macroblock access frequency is kept in an n-bit counter (MAT). Reducing the counter size in MAT generally leads to decreased performance [10]. However, keeping tour history using a 2- or 3-bit counter in NTS and PCS showed virtually no performance benefit over the 1-bit scheme.

6 Using Reuse Information in Data Cache Management

Each of the multi-lateral schemes operates on the assumption that reuse information is useful in actively managing the cache. In this section, we assess the value of reuse information for making placement decisions in multi-lateral L1 cache structures. We first examine optimal cache structures to determine how they exploit reuse information in cache management. We then outline the experiments performed to validate the use of reuse information and compare the performance of multi-lateral schemes to the performance of near-optimally managed caches of the same size.

6.1 Optimally and Near-Optimally Managed Caches

Belady's MIN [2] is an optimal replacement algorithm for a single-structure cache, i.e. it results in the fewest misses[5]. While it is interesting to see how reuse information is used

[5]Note that we do not consider timing models in this section, for which MIN is not an optimal algorithm.

to manage a single cache, we are interested in determining how an optimal replacement algorithm for a multi-lateral cache makes replacement decisions, and how reuse information might best be exploited. However, no direct extension of MIN to multi-lateral caches is known. The only exception is where both the A and B caches of a multi-lateral configuration are fully associative and blocks are free to move between the two caches as necessary in order to retain the most useful blocks in the cache structure; this multi-lateral cache is, however, degenerate as it reduces to a single fully associative cache of size equal to the total of cache A plus cache B. In this case, MIN can be used to optimally manage the hardware-partitioned fully associative single cache.

We refer to Belady's MIN algorithm, when applied to the dual-fully associative caches, as *opt*. While *opt* gives an upper bound on the performance of a multi-lateral cache of a given size and associativity, comparing *opt* to the implementable schemes does not yield a direct comparison of replacement decisions based on reuse information. Since multi-lateral caches typically have caches A and B of differing associativity, the performance difference between the implementable schemes and *opt* may be due not only to replacement decisions, but also to mapping restrictions placed on the implementable schemes by their limited associativity. We would instead like to compare the performance of the implementable schemes to an optimally managed multi-lateral configuration where the A and B caches are of differing associativity in order to better attribute the differences in placement and replacement decisions to the management policy itself, rather than to the associativity of the configuration.

Pseudo-opt [20] is a multi-lateral cache management scheme for configurations where the associativity of A is no greater than that of B. As in the configuration for *opt*, free movement of blocks between A and B is allowed in this scheme, provided that the contents of A and B are disjoint. Management is adapted from Belady's MIN algorithm, as follows. On a miss, the incoming block fills an empty slot in the corresponding set of cache A or cache B, if one exists. If no such empty slot exists, then for each set of cache A, an extended set of blocks is defined, consisting of all blocks in cache A and cache B that map to this set in cache A.

Reference	b	c	F	D	b	E	D	F	c
Time	1	2	3	4	5	6	7	8	9
Set 0	–	–	F	F	F	D	D	D	D
Set 1	b	b	b	b	b	b	b	b	b
Cache B		c	c	D	D	E	E	F	c
Hit/Miss	***M***	***M***	***M***	***M***	***H***	***M***	***H***	***M***	***M***

Figure 4: An example showing why *pseudo-opt* is suboptimal.

For each extended set that includes any block currently resident in cache B, i.e. sets whose extended set is larger than the associativity of cache A, block γ of the extended set whose next reference is farthest in the future is found. If block γ does not currently reside in cache B, it is swapped with one of the cache B blocks of this extended set. The incoming block is placed in cache A and the block in that set next referenced farthest in the future is moved to cache B, overfilling it by one block. The cache B block next referenced farthest in the future is then replaced. This choice is not optimal in all cases, as illustrated by the following example.

For the reference pattern in Figure 4, consider a design with a direct-mapped cache A of size 2 blocks (2 sets) and a one block cache B. The references shown in the figure are block addresses. Upper case letters map to set 0 and lower case letters map to set 1 of cache A. The figure shows the contents of set 0 and set 1 in cache A and cache B after each memory access. Using *pseudo-opt* we incur 7 misses. The first 3 compulsory misses fill empty blocks. **D** replaces **c** at time 4 since **c** is next referenced further in the future than **F** or **b**. At time 6, **E** cannot replace **b** and replaces **F** rather than **D**. Finally, **F** and **c** miss. However, the minimum possible number of misses is 6. This can be achieved by replacing **F** (in set 0 of cache A) instead of **c** (in cache B) at time 4. **E** then replaces **b** at time 6 after swapping **b** and **c**, and finally **F** misses, but **c** hits.

In cases where the A and B caches are fully associative, *pseudo-opt* reduces to *opt*. Although *pseudo-opt* is also not an implementable policy, its performance, as seen in Table 6, is close to that of *opt*. Furthermore, *pseudo-opt*'s performance is much better than the implementable multi-lateral schemes' performance. Most of the performance difference between the implementable schemes and *opt* is thus due to non-optimal allocation and replacement decisions; only a small portion of the performance difference (no more than the difference between *opt* and *pseudo-opt*) is due to the restricted associativity of the implementable schemes. We therefore use *pseudo-opt* for comparison to the implementable schemes in order to eliminate the associativity difference from the evaluations and give a better idea of the realizable performance of a limited-associativity multi-lateral cache. The differences in placement and replacement decisions seen in the implementable schemes provide insights into their performance relative to a near-optimal scheme.

6.2 Simulation Environment

To evaluate the *opt* and *pseudo-opt* schemes, we collected memory reference traces generated from the *SimpleScalar* processor environment [3]. Each trace entry contains the (effective) address accessed, the type of access (load or store), and the program counter of the instruction responsible for the access.

We skipped the first 100 million instructions (to avoid initialization effects) and analyzed the subsequent 25 million memory references. We limited the number of memory references evaluated due to the space and processing time required to perform the *opt* and *pseudo-opt* cache evaluations. These experiments use five SPEC95 integer benchmarks – **compress**, **gcc**, **go**, **li**, and **perl**, since sampling such a small portion of a floating point program will likely generate references that form part of a regular loop, resulting in very low miss rates. However, the sampled traces for the integer programs do reasonably mirror the actual memory reference behavior of the complete program execution, as shown in Section 5.

The traces were annotated to include the information necessary to perform the *opt* and *pseudo-opt* replacement decisions and counters for many useful statistics. These included the outcome of an access (hit or miss) as it would have occurred in an *opt*/*pseudo-opt* configuration, the usage information for each block tour (as seen for each scheme), and the number of blocks in each reuse category that are in the cache at each instant in time (i.e. the number of NTNS, NTS, TNS, and TS blocks resident in the cache). From these statistics, we gathered information regarding the performance of the *opt* and *pseudo-opt* management schemes and compared the performance of the implementable schemes to that of the optimal schemes on an access-by-access basis.

Due to the smaller size of the input sets in this section, compared to the full program executions performed in Section 4, we chose a direct-mapped, 8KB A cache and a fully associative, 1KB B cache, each with a 32B blocksize, for the *pseudo-opt* configuration. The *opt* configuration is simply a 9K fully associative cache. Using the larger, (16+2)K caches of Section 4 for these evaluations would not have been useful in determining each scheme's performance, as the SPEC benchmarks already have small to moderately sized working sets [5]; these relatively short traces would show little performance benefit from active management in a large cache, whereas the benefits of active management are highlighted when using smaller caches. The *mlcache* simulator used in this section only deals with the cache and memory, not the processor. Without processor effects, timing is of little significance and *mlcache* is used here only as a behavioral-level simulator.

6.3 Results

6.3.1 Analysis of *opt* vs. *pseudo-opt*

We analyzed the annotated traces produced from the *opt* and *pseudo-opt* runs to determine their relative performance based on miss ratio measurement and block usage information. In particular, we counted the number of tours that show each reuse pattern, the number of each type of block resident in the cache at any given instant, and how often a block with a prior reuse characteristic changes its usage pattern in a subsequent tour.

6.3.2 Miss Ratio

As the miss ratios in Table 6 show, the performance of *pseudo-opt* is relatively close to that of *opt*, except for **go**. Note that miss ratio in these experiments is a straightforward performance metric, as the simulations are behavioral and do not include access latencies or processor latency-masking effects. In **go**, the performance disparity between *opt* and *pseudo-opt* is due to the limitations on the associativity of the A cache, and possibly also

	Compress	Gcc	Go	Li	Perl
opt	0.042	0.013	0.008	0.008	0.0169
pseudo-opt	0.048	0.016	0.023	0.010	0.0174
PONS	0.050	0.018	0.028	0.011	0.0175
NTS	0.063	0.029	0.054	0.018	0.0200
PCS	0.067	0.032	0.060	0.019	0.0190
16K:1w	0.063	0.029	0.050	0.020	0.0520

Table 6: Miss ratios for the five (8+1)KB and the 16KB cache configurations on the trace inputs. *PONS* is the *pseudo-opt no-swap* scheme.

to the suboptimal replacement policy of *pseudo-opt*. On each replacement, any number of swaps can be done in *opt* to rearrange the cache contents so that the least desired cache block is replaced. However, in *pseudo-opt*, the movement choices are limited by the mapping requirements of the direct-mapped A cache – at most one block, mapping to the set in B that is associated with the incoming block, need be swapped on each replacement. Despite its more limited choice of blocks to replace, the performance of *pseudo-opt* is still very close to *opt* except for **go**, and the difference in performance between *pseudo-opt* and *opt* is always much smaller than the difference between the implementable configurations and *pseudo-opt*. The actual performance of the implementable schemes is discussed in Section 6.4.

In addition to the advantage of future knowledge, *pseudo-opt* differs from the implementable schemes by freely allowing blocks to move between the A and B caches in order to obtain the best block for replacement. However, this movement actually accounts for very little of the performance difference seen between *pseudo-opt* and the implementable schemes.

To verify this claim, we created a version of *pseudo-opt*, called *PONS* (*pseudo-opt no-swap*), that disallows data movement between the caches. The management scheme used by *PONS* is the same as for *pseudo-opt*, except that no blocks are swapped between cache A and B at any time. The incoming block replaces the block in its A or B set that is next referenced farthest in the future. In *PONS*, it is thus possible that a replaced block in cache B is referenced sooner in the future than a block in some other set in A that extends into that set of cache B and could have been replaced if swaps were allowed.

However, we see in Table 6 that the miss ratios of *pseudo-opt* and *PONS* are actually

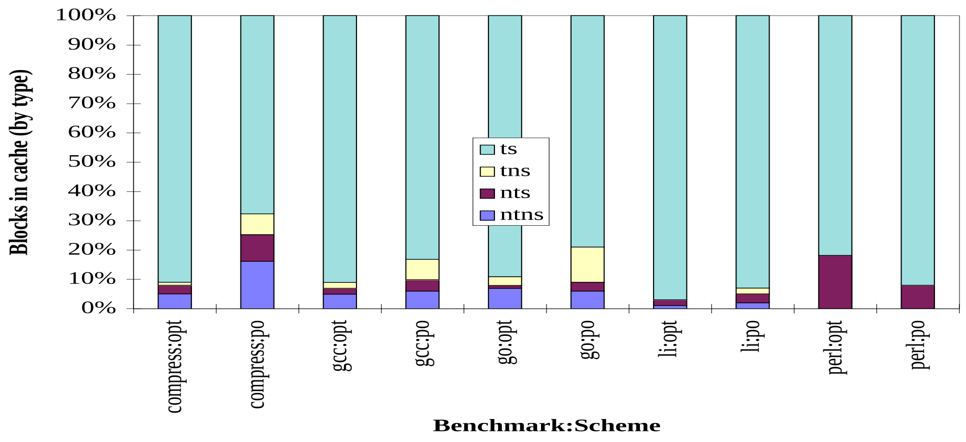


Figure 5: Dynamic cache block occupation in *opt* and *pseudo-opt* (denoted *po*, grouped into NTNS, NTS, TNS, and TS usage patterns.

very close, indicating that the omission of inter-cache data movement has a small effect on near-optimum performance. In particular, the performance difference between *pseudo-opt* and *PONS* is much smaller than the difference between *pseudo-opt* and the implementable schemes, showing that the major advantage of *pseudo-opt* comes from its management using future knowledge as opposed to its ability to move blocks between caches. Since some multi-lateral schemes do allow data movement between the A and B caches, we use *pseudo-opt* as the basis for comparison to the implementable multi-lateral cache schemes.

6.3.3 Cache Block Locality Analysis

We examined the locality of cache blocks that are resident in the cache structure at any given time by counting the number of blocks in each category at the time of each miss and taking an average over the duration of the program. The locality of the cache blocks for the *opt* and *pseudo-opt* configurations is shown in Figure 5.

For *opt* and *pseudo-opt* managed caches, as expected, data that exhibits temporal reuse occupies a large portion of the cache space; nontemporal data occupies at most 23% of the cache (**compress** under *pseudo-opt*). Furthermore, the vast majority of the blocks in L1 are both temporal *and* spatial. Though the blocks are relatively small in size (32 bytes), we find that spatial reuse can be exploited well if the cache is managed properly. As expected, the *pseudo-opt* configuration (vs. the *opt* configuration) generally holds fewer TS blocks in

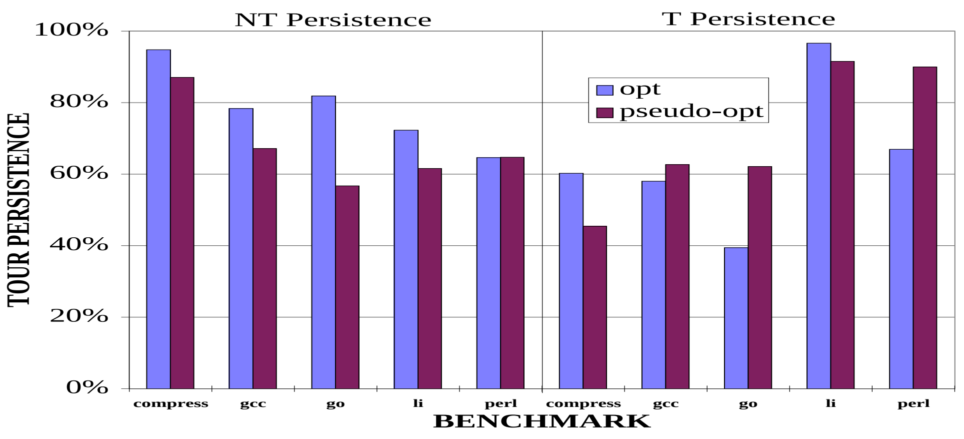


Figure 6: Block usage persistence within *opt* and *pseudo-opt*.

the cache, due to the *pseudo-opt* configuration's limited placement options and sub-optimal replacement decisions. However, **perl** is an exception to this observation. In both *opt* and *pseudo-opt*, all of **perl**'s data in the cache is spatial (TS or NTS), as seen in Figure 5; keeping an NTS block in cache long enough so that it obtains TS status, as done by *pseudo-opt*, slightly increases the miss ratio of the **perl** benchmark above that of *opt*, indicating that maximizing the number of TS blocks in cache is not always the best policy.

6.3.4 Block Usage Persistence

While the presence of certain types of blocks in the cache shows the potential benefit of managing the cache with reuse information, management based on this information is not straightforward. Blocks can have different usage characteristics during different portions of program execution, making block usage hard to predict. The prediction of a particular block's usage pattern is similar to the branch prediction problem. However, branch outcomes are easier to predict than optimal block usage characteristics for aiding placement decisions.

To assess the value of reuse information, we examined the *persistence* of cache block behavior in the *opt* and *pseudo-opt* schemes, i.e. once a block exhibits a given usage characteristic in a current tour, how likely is it to maintain that characteristic in its next tour? If there is a high correlation between past and future use (i.e. the block's usage characteristic is persistent), prediction of future usage behavior will be easier. Block *persistence* in terms of *same* and *not-same* is therefore analogous to the *same-direction* terminology used in branch prediction studies [14] to predict the path a specific branch will take given behavior history. Instead of determining block persistence in terms of the four usage patterns NTNS, NTS, TNS, and TS, we decided to examine only the persistence of T and NT patterns. This coarser granularity grouping is more directly relevant to our placement decisions for 2-unit cache structures.

Figure 6 presents data for block usage persistence in successive tours. In general, blocks that exhibit NT usage behavior in prior tours have a strong likelihood of exhibiting NT behavior again in future tours. In the *opt* scheme, this likelihood ranges from 63% (**perl**) to 95% (**compress**) for the evaluated benchmarks. However, the *pseudo-opt* scheme shows somewhat less persistence, ranging from 57% (**go**) to 87% (**compress**) for NT blocks. Overall, the persistence of NT blocks in both *opt* and *pseudo-opt* schemes is well over 50%. Blocks that exhibit T usage behavior are less persistent, and thus less predictable. T block persistence in the *opt* scheme ranges from 40% (**go**) to 97% (**li**), and is similar for *pseudo-opt*: 45% (**compress**) to 92% (**li**).

li skews these numbers somewhat; regardless of the type of usage characteristics that a block exhibited in a tour, its next tour is highly likely to exhibit temporal reuse. While temporal blocks are persistent in **li**, they are much less persistent for the other three benchmarks. As **compress**, **gcc**, **go**, and **perl** represent a wider range of program execution than **li** alone, we see that future tours of temporal blocks are harder to predict than nontemporal blocks, i.e. temporally tagged blocks are only weakly biased toward exhibiting T usage patterns in their next tour.

6.4 Multi-Lateral Scheme Performance

Given the performance and reuse information of the *opt* and *pseudo-opt* configurations, we can determine how the implementable schemes perform as a result. We restrict our evaluation of the implementable schemes to NTS and PCS, the two multi-lateral configurations that actively place data within L1 based on individual block reuse information.

6.4.1 Miss Ratio Performance of NTS and PCS

To compare with the *pseudo-opt* configuration used, we configured the NTS and PCS structures to have an 8KB direct-mapped A cache, a 1KB fully associative, LRU-managed B cache, 32B block size, and a 32-entry detection unit (DU).

The miss ratio performance of the two configurations is shown in Table 6 along with the performance of *opt*, *pseudo-opt*, *PONS*, and a direct-mapped 16KB single structure cache. In concurrence with results of earlier analyses of these multi-lateral configurations [16][17][22], the NTS and PCS caches each perform about as well as a direct-mapped cache of nearly twice the size. However, as the performance of *pseudo-opt* indicates, there is still room for further improvement.

6.4.2 Prediction Accuracy

NTS makes placement decisions based on a given block's usage during its most recent past tour; if the block exhibited nontemporal reuse, it is placed in the smaller B cache for its next tour, otherwise it is placed in the A cache. PCS makes placement decisions based on the reuse of blocks loaded by the memory accessing instruction. If the most recently replaced block loaded by a particular PC exhibited nontemporal reuse, the next block loaded by that PC is predicted to do the same and is placed in the smaller B cache, otherwise it is placed in the larger A cache. In both schemes, accessed blocks with no matching entry in the DU are placed in the A cache by default.

The accuracy of these predictions can be determined based on the actual usage of the blocks in the *pseudo-opt* scheme. For instance, a prediction of T behavior for a block is classified as correct if the actual usage of that block in the *pseudo-opt* scheme is T. As noted in Section 6.2, the annotated traces fed to our trace-driven cache simulator, a modified version of *mlcache*, contain the actual block usage information for the tours seen in the *pseudo-opt* scheme. The simulator provides information on the selected scheme's block prediction and actual usage accuracy. Given this information, it is easy to determine how well each of the configurations predict a block's usage, and consequently, whether it is properly placed within

		Tour prediction accuracy		Actual usage accuracy	
		NTS	PCS	NTS	PCS
Compress	**NT**	0.483	0.703	0.859	0.828
	T	0.240	0.210	0.402	0.412
	Total	0.247	0.495	0.725	0.725
Gcc	**NT**	0.322	0.589	0.684	0.617
	T	0.539	0.596	0.703	0.704
	Total	0.527	0.595	0.696	0.675
Go	**NT**	0.337	0.454	0.573	0.519
	T	0.588	0.621	0.646	0.652
	Total	0.561	0.596	0.624	0.609
Li	**NT**	0.225	0.243	0.390	0.363
	T	0.815	0.815	0.873	0.866
	Total	0.786	0.723	0.787	0.758
Perl	**NT**	0.406	0.817	0.934	0.950
	T	0.326	0.382	0.727	0.738
	Total	0.327	0.510	0.861	0.875

Table 7: Tour prediction and actual usage accuracy for NTS and PCS, broken into NT, T, and overall (total) accuracy. Accuracies are relative to actual block usage in *pseudo-opt*.

the L1 cache structure.

Table 7 shows the prediction accuracy of NTS and PCS for the benchmarks examined. In general, ignoring **li** due to its excessively high temporal reuse, the prediction accuracy is relatively low, ranging from 25% (**compress** with NTS) to 60% (**go** with PCS), despite the larger granularity of block typing (two categories, T and NT, rather than the complete four category breakdown). Both the NTS and PCS schemes show poor prediction accuracies, directly impacting their block allocation decisions and their resulting overall performance. Improved block usage prediction for these schemes may result in better block placement and higher performance.

6.4.3 Actual Usage Accuracy

Regardless of the prediction accuracy, a given block will exhibit reuse characteristics based on the duration and time of its tour through the cache. We examined the actual usage of each block as it was evicted from each of the caches in the implementable schemes to see how that usage compared to the same blocks usage in the *pseudo-opt* scheme. This comparison sheds some light on the effect of eliminating the movement of blocks between the caches during an L1 tour.

	Compress		Gcc		Go		Li		Perl	
	NT	T	NT	T	NT	T	NT	T	NT	T
pseudo-opt	1293.3	26695	1731.2	25946	1177.8	22297.3	4400.4	34221.5	957	54749
NTS	4.1	44.1	2.0	52.0	1.4	26.1	3.3	66.3	7.7	130.4
PCS	3.7	48.9	2.1	49.8	1.3	24.1	3.2	66.1	7.9	134.0
8K:1w	3.6	35.4	2.0	35.7	1.2	14.0	2.6	48.0	3.7	35.9

Table 8: Average tour lengths for *pseudo-opt*, NTS, PCS, and a direct-mapped single cache. Tour lengths measure the number of accesses handled while the block is in cache. Tours are broken into NT and T.

We see in Table 7 that relative to the low prediction accuracy we saw in Section 6.4.2, the actual usage of the blocks in the cache is closer to the actual usage of the blocks in *pseudo-opt*. Thus, despite our placement decisions, some blocks still exhibited reuse behavior akin to that seen in the *pseudo-opt* configuration. Furthermore, although NTS exhibited lower tour prediction accuracy than PCS (in all the benchmarks except for **li**), it exhibited higher actual usage accuracy (in all the benchmarks except for **perl**).

6.4.4 Tour Lengths

While these accuracies are interesting, the block tour lengths in the implementable schemes are not directly related to the tour lengths in the *pseudo-opt* scheme, as *pseudo-opt* has a more elaborate replacement policy. Disparate tour lengths can affect these comparisons in two ways. First, for tours that are shorter in the *pseudo-opt* configuration than in the implementable schemes, the implementable schemes may keep a block in L1 longer than necessary and permit it to exhibit seemingly more beneficial usage patterns (i.e. TNS or TS). While this makes a particular block seem more useful, the longer presence of that block in L1 in the implementable schemes may unduly shorten the tours of a significant number of other blocks, leading to their misclassification and precluding their optimal placement.

Conversely, where tours are longer in *pseudo-opt* than in the implementable schemes, the corresponding blocks in the implementable schemes may be replaced before they can exhibit their optimal usage characteristics, causing poor usage accuracy, poor prediction accuracy, and poor placement into the small B cache, and hence shorter tour lengths.

Table 8 shows the average tour length for blocks showing T and NT usage characteristics. As expected, NT tours are much shorter than T tours, and tours in *pseudo-opt* are on average much longer than tours in either NTS or PCS, due to its future knowledge and greater flexibility in management of data once it has entered the L1 cache structure. In NTS and PCS, once a block is placed in L1, it remains in the same cache until it is replaced, and is thus subject to the replacement policy inherent in the corresponding cache. For instance, if block α is deemed to be T in either NTS or PCS, it is placed in the direct-mapped A cache. If a subsequent block β maps to the same set in A and is also marked T, the earlier T block, α, will be evicted, possibly before it can exhibit its optimum usage characteristics. In *pseudo-opt*, if it is deemed desirable at that time, block α would simply be moved to the B cache. As a result, the tour length of block α would tend to be much longer under *pseudo-opt* than under NTS or PCS.

We see that there is a clear gap between the average tour length of blocks in *pseudo-opt* vs. the implementable schemes. This difference in tour lengths is, however, much larger than the difference in miss ratio for the implementable vs. *pseudo-opt* schemes. The performance difference is much smaller because the average tour length in *pseudo-opt* has increased greatly, but the tour lengths of conflicting blocks may not be helped as much by the better management scheme. Some blocks may be chosen to remain in cache for nearly the entire program's execution, as they are accessed regularly (though not necessarily frequently), and would thus have very long tour lengths. These long-lived blocks greatly increase the average tour lengths seen for each benchmark, though they may only reduce the overall number of misses by a small amount.

From this study we see that by improving upon the prediction of the usage characteristics of a block and the management of those blocks once they are placed in the L1 cache structure, we might improve the performance of the NTS and PCS schemes to reduce the performance gap that exists between these schemes and *pseudo-opt*. We see from Table 8 that each of the multi-lateral schemes does increase the tour lengths relative to the single, direct-mapped cache structure of nearly the same size, indicating that these schemes are making decisions that improve data usage and performance relative to a conventional cache.

7 Conclusions

In this paper we have evaluated three different implementable methodologies (MAT, NTS, and PCS) for managing an on-chip data cache based on active block allocation via capturing and exploiting reuse information. In general, an actively managed cache structure significantly improves upon the performance of a traditional, passively-managed cache structure of similar size and competes with one of nearly twice the size. Further, the individual effective address reuse history scheme used in NTS generally gives better performance than the macroblock effective address-based MAT or the PC-based PCS approaches.

We compared the performance of the PCS and NTS schemes to the performance of a near-optimally managed cache structure (*pseudo-opt*). The difference in performance, block usage prediction, actual block usage, and tour lengths between the implementable schemes and *pseudo-opt* shows much room for improvement for these actively-managed caches.

Thus, multi-lateral cache structures that actively place data within the cache show promise of improving cache space usage. However, the prediction strategies used in these current schemes are too simple. Improving the prediction algorithms, as well as actively managing blocks once they are placed in the L1 cache structure (active replacement), can help improve the performance of the implementable schemes and may enable them to approach optimal cache space usage.

8 Acknowledgments

This research was supported by the National Science Foundation grant MIP 9734023 and by a gift from IBM. The simulation facility was provided through an Intel Technology for Education 2000 grant.

References

[1] J.-L. Baer and T.-F. Chen. An effective on-chip preloading scheme to reduce data access penalty. In *Proceedings of Supercomputing'91*, pages 176–186, November 1991.

[2] L. A. Belady. A study of replacement algorithms for a virtual storage computer. In *IBM Systems Journal, Vol. 5*, pages 78–101, 1966.

[3] D. Burger and T. M. Austin. Evaluating future microprocessors: the simplescalar tool set. In *University of Wisconsin – Madison Technical Report #1342*, 1997.

[4] D. Callahan, K. Kennedy, and A. Porterfield. Software prefetching. In *Proceedings of ASPLOS-IV*, pages 40–52, 1991.

[5] M. J. Charney and T. R. Puzak. Prefetching and memory system behavior of the spec95 benchmark suite. In *IBM Journal of Research and Development, Vol. 41 Number 3*, pages 265–286, May 1997.

[6] T.-F. Chen and J.-L. Baer. Reducing memory latency via non-blocking and prefetching caches. In *Proceedings of ASPLOS-V*, pages 51–61, 1992.

[7] C-H. Chi and H. Deitz. Improving cache performance by selective cache bypass. In *Proceedings of 22nd Hawaii International Conference on System Science*, pages 277–285, January 1989.

[8] A. Gonzalez, C. Aliagas, and M. Valero. Data cache with multiple caching strategies tuned to different types of locality. In *Proceedings of the 1995 Conference on Supercomputing*, pages 338–347, 1995.

[9] M. D. Hill. Dineroiii documentation. In *University of California – Berkeley Unpublished UNIX-style Man Page*, October 1985.

[10] T. Johnson and W. W. Hwu. Run-time adaptive cache hierarchy management via reference analysis. In *Proceedings of ISCA-24*, pages 315–326, 1997.

[11] N. P. Jouppi. Improving direct-mapped cache performance by the addition of a small fully associative cache and prefetch buffers. In *Proceedings of ISCA-17*, pages 364–373, 1990.

[12] D. Kroft. Lockup-free instruction fetch/prefetch cache organization. In *Proceedings of ISCA-8*, pages 81–85, May 1981.

[13] G. Kurpanek, K. Chan, J. Zheng, E. DeLano, and W. Bryg. Pa-7200: A pa-risc processor with integrated high performance mp bus interface. In *COMPCON Digest of Papers*, pages 375–382, February 1994.

[14] C-C. Lee, I-C. K. Chen, and T. N. Mudge. The bi-mode branch predictor. In *Proceedings of MICRO-30*, pages 4–13, December 1997.

[15] V. Milutinovic, A. Milenkovic, and G. Sheaffer. The cache injection/cofetch architecture: Initial performance analysis. In *Proceedings of MASCOTS'97*, pages 63–64, January 1997.

[16] J. A. Rivers and E. S. Davidson. Reducing conflicts in direct-mapped caches with a temporality-based design. In *Proceedings of the 1996 ICPP*, pages 154–163, August 1996.

[17] J. A. Rivers, E. S. Tam, and E. S. Davidson. On effective data supply for multi-issue processors. In *Proceedings of ICCD'97*, pages 519–528, October 1997.

[18] J. A. Rivers, E. S. Tam, G. S. Tyson, E. S. Davidson, and M. Farrens. Utilizing reuse information in data cache management. In *Proceedings of ICS'98*, pages 449–456, 1998.

[19] G. S. Sohi and M. Franklin. High-bandwidth data memory systems for superscalar processors. In *Proceedings of ASPLOS-IV*, pages 53–62, 1991.

[20] Vijayalakshmi Srinivasan and E. S. Davidson. Improving performance of an l1 cache with an associated buffer. In *University of Michigan – Ann Arbor Technical Report CSE-TR-361-98*, March 1998.

[21] E. S. Tam and E. S. Davidson. Early design cycle timing simulation of caches. In *University of Michigan – Ann Arbor Technical Report CSE-TR-317-96*, November 1996.

[22] E. S. Tam, J. A. Rivers, G. S. Tyson, and E. S. Davidson. *mlcache*: A flexible multi-lateral cache simulator. In *Proceedings of MASCOTS'98*, pages 19–26, 1998.

[23] G. Tyson, M. Farrens, J. Matthews, and A. R. Pleszkun. A modified approach to data cache management. In *Proceedings of MICRO-28*, pages 93–103, December 1995.