Predictive Sequential Associative Cache

Brad Calder   Dirk Grunwald
Department of Computer Science,
University of Colorado
Campus Box 430
Boulder, CO 80302-0430
calder,grunwald@cs.colorado.edu

Joel Emer
Digital Semiconductor,
77 Reed Road (HLO2-3/J3) Hudson, MA 01749
emer@vssad.hlo.dec.com

December 5, 1994

Abstract

Traditionally, set-associative caches are implemented by comparing all blocks in a cache set in parallel for each reference and then selecting the desired block from the set. By providing more than one location for holding the data for a particular memory address, set associativity reduces the cache miss rate for most programs. The traditional solution is, however, not without cost. As contrasted with direct-mapped caches, values from a set-associative cache can not be used by the processor, even speculatively, until all the comparisons complete. This results in a greater access time than comparably sized direct-mapped caches. Fortunately, however, in most programs, cache references are to the most recently used items. A number of researchers have proposed cache designs that exploit this locality to reduce access time and miss rate.

In this paper, we propose a cache design that provides the same miss rate as a two-way set associative cache, but with a access time closer to a direct-mapped cache. As with other designs, a traditional direct-mapped cache is conceptually partitioned into multiple banks, and the blocks in each set are probed, or examined, sequentially. Other designs either probe the set in a fixed order or add extra delay in the access path for all accesses. We use *prediction sources* to guide the cache examination, reducing the amount of searching and thus the average access latency. A variety of accurate prediction sources are considered, with some being available in early pipeline stages.

1 Introduction

Set-associative caches offer lower miss rates than direct-mapped caches, but usually have a longer access time. In a set-associative cache, cache blocks are divided into sets and a single fetch address may find its data in any block of its set. When referencing a cache set, the blocks are examined in parallel. When all blocks and, in specific, their tags have been read from the set, a tag comparator selects the block containing the desired data, and passes the data from that block to the processor. The comparator introduces an extra delay between the availability of the cache data and the time it can be used by the processor, because the cache can not send the data to the processor, even speculatively, until the comparison is finished. This, among other factors, increases the cache access or cycle time. By contrast, in a direct-mapped cache, a fetch address can only access a single block, and the data from that block can be speculatively dispatched while the tags are compared, thus reducing the critical path length. The CACTI cache timing simulator [13] can be used to approximate the increased access time for a two-way set-associative cache. For caches with 32 byte cache lines, the same configuration used in our simulation study, the access time for a two-way associative cache is 1.51, 1.46 and 1.40 times longer than the access time for a direct mapped cache for 8KB, 16KB and 32KB caches, respectively.

The design tradeoff between miss rate and access time in set-associative caches versus direct mapped caches has led several researchers to suggest ways to achieve miss rates similar to two-way set-associative caches with fast access times by using modified direct-mapped caches. Recently, several researchers have suggested using a serialized or sequential search to compare tags in set-associative caches. In some of these schemes, the cache is divided into sets with 2 blocks per set. When accessing a cache set, first one block is probed. If a match is found, processing continues; if not, the second block in the set is probed. If no match is found, a miss occurs. If the cache line is usually found in the first block to be examined, the average access time will be less than a direct-mapped cache (because the miss rate would be similar to a two-way set-associative cache) or a two-way set-associative cache (because the cycle time is that of a faster direct-mapped cache).

Several variants have been proposed. In this paper, we compare and extend several proposed schemes for implementing two-way set-associative caches at the first level of the memory hierarchy. Our technique uses a number of *prediction sources* to pick the first block to probe. These predictors include information about the source or destination of a fetch, the instruction or procedure fetching the data and the effective fetch address. Our technique offers the low miss rate of a two-way set-associative cache and the low cycle time of a direct-mapped cache; the performance varies with the accuracy of the prediction information. The source for prediction information can be adjusted for pipeline constraints; implementors can trade lower prediction rates for the freedom to reduce timing constraints. In this paper, we examine only two-way set-associative caches; while higher degrees of associativity can be implemented using these techniques, there are fewer advantages at higher associativities.

The contributions of this paper are:

1. We extend sequential cache designs to use prediction sources, allowing sequential two-way set-associative caches to be used for first-level caches. Our cache design is simpler to implement than previous designs and more effective.
2. Describe several prediction sources and their efficacy.
3. Present a performance comparison that distinguishes between access latency and *cache occupancy*, or the time the cache is busy during memory references.

2 Prior Work

Sequential or serial implementations of two-way set-associative caches can be categorized along two dimensions. First, they can be categorized by whether the cache is probed in a statically fixed order, or if it is probed in a dynamically determined order. Second, they can be categorized by the block allocation (and re-allocation) policy. The addition of a block re-allocation policy is especially important in those schemes with a fixed probe order.

We will use the diagrams in Figure 1 to clarify issues in the unconventional implementations. We assume the reader is familiar with the design of direct-mapped caches and the conventional implementation of two-way set-associative caches that use content-addressable memory. We assume each cache entry has an L-bit *set index* and a *tag*. We use a word-addressed cache to describe the various cache organizations to simplify their description. A direct-mapped cache contains 2^L cache blocks; when a block address is to be fetched or loaded into the cache, the lower L bits are used to index into the cache. When fetching an item from the cache, the remaining address bits are compared to the tag; if these match, the item is successfully found in the cache, otherwise a cache miss occurs. In a two-way set-associative cache, only $L - 1$ bits are used as a set index. Each cache set contains two *blocks*. In a conventional design, the tags in both blocks are compared concurrently when fetching an address. If either tag matches, the reference has been found. New entries may be loaded into either block, as determined by the block allocation policy. Usually a least-recently-used mechanism is employed. A set-associative cache typically has a lower miss rate than a direct-mapped cache, but is more difficult to implement and increases the cache access time [5].

2.1 Statically Ordered Cache Probes

Agarwal *et al* [1] proposed the Hash-Rehash cache (HR-Cache) to reduce the miss rate of direct-mapped caches. Although the Hash-Rehash cache can be used to implement arbitrary associativity, we only consider a two-way set-associative cache, since that is the most cost-effective configuration. The HR-cache organization uses a fixed probe order and a non-LRU cache allocation/re-allocation scheme to maximize the number of first probes that hit.

Figure 1(a) illustrates the structure of the HR-cache. The cache blocks are divided into two banks. Each bank contains one of the two blocks that comprise a set, and $L - 1$ bits are used to index a block in a set. While a conventional set-associative cache matches the tags in parallel, the HR-cache matches the tags *sequentially.* The cache is organized as a conventional direct-mapped cache; thus, the cycle time for accessing an individual block would be faster than that of a conventional set-associative cache. Cache blocks are indexed using two distinct hashing functions, and the cache is probed in a fixed order using those hash functions. We call the cache block addressed by the first hash function the "first block" and the remaining block the "second block." If the data is not found in the first block, the second block is examined. If the data is found in the second block, the contents of the first and second blocks are exchanged. The intuition is that by re-allocating the data, successive references will locate the data quicker, because it will be found with the first probe. If the data is not found by the second probe, a miss occurs. The first block is moved to the second block, since the first block was more likely to have been more recently referenced than the second block, and the missing block is placed in the first block. As shown in later examples, this mechanism does not implement a true LRU replacement policy.

This process is easier to understand by example. We will use the same reference stream to illustrate the operation of each cache organization. The references stream is:

1^{st} reference	010_10
2^{nd} reference	001_10

The addresses are specified in binary, and the tag (the high three bits) is separated from the set index (the low two bits) by an underscore (_). Using the cache configuration in Figure 1(a), all of these references map to the same set.

Since the HR-cache uses a conventional direct-mapped organization, the concept of blocks in a set are represented by the use of an additional cache index bit to distinguish the two blocks of the set. Before each example, we assume that address 000_10 (at index two) and 001_10 (at index six) are in the two blocks of the set, and that 001_10 was more recently referenced than 000_10.

In the HR-Cache the first hashing function uses a direct-mapped lookup (formed using the lower-order bits of the tag, concatenated with the set index), and the second hashing function probes the other block in the set Thus, fetching the address 010_10 would first probe index two, then index six, while fetching 001_10 would first probe index six and then index two. Figure 1(a) illustrates how the two references are processed in the HR-Cache. The fetch of 010_10 would examine index two first, and index six second. Since a miss occurs, the contents of index two (000_10) are moved to index six, and 010_10 is loaded into index two. Notice that block 001_10 is replaced, even though it was more recently referenced than 000_10. A true LRU replacement strategy would have replaced 000_10. The next reference is to 001_10, which was just replaced. Index six is examined first, and index two is examined second. Again, a miss occurs. The contents of index six are moved to index two, and 001_10 is loaded into index six.

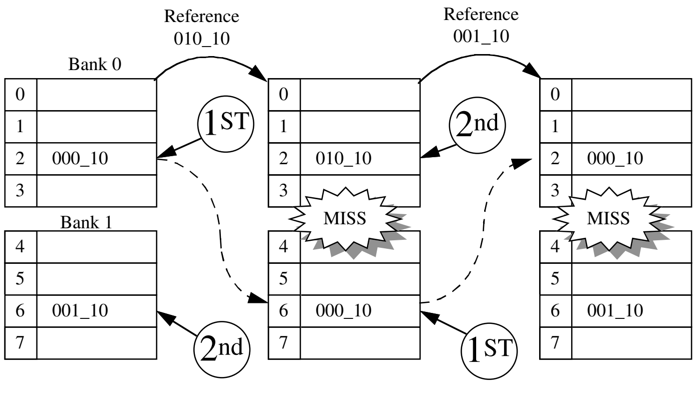
Cache simulations by Agarwal [2], and our own simulations, show that the Hash-Rehash cache has a higher miss rate than a two-way set-associative cache with LRU replacement. There are two obstacles that limit the performance or practicality of the HR-Cache: the need to exchange entire cache lines and the high miss rate. In the HR-Cache, the entire first and second blocks must be exchanged if a reference is found in the second block. This may be a problem for caches with large lines. Some architectures use a 128-byte cache line, although the processor only retrieves four or eight bytes on each fetch. Exchanging such large cache lines either requires a several cycle exchange operation, or a design that allows two cache lines to be rapidly exchanged via a wide cache access. The extra metal and sense amplifiers required for this wide access would likely add significantly to both the size and power requirements of the cache. Furthermore, a special exchange operation could interfere with the pipelining of normal cache accesses, especially in multi-ported designs, and would introduce more bookkeeping for deferred write and fill operations.

The Column-Associative Cache (CA-Cache) of Agarwal and Pudar [2] improved the miss rate and average access time of the HR-Cache, but still requires that entire cache blocks be exchanged. Like the HR-Cache, the CA-Cache, shown diagrammatically in Figure 1(b), divides the cache into two banks. Cache blocks are found using two hashing functions. In the CA-Cache, a rehash bit is associated with each cache block, indicating that the data stored in that cache block would be found using the second hash function. The rehash bit is the exclusive-or of the lowest-order bit in the tag and the bank number. Consider the process of fetching the addresses 010_10 and 001_10. As before, 000_10 and 001_10 are already at indices two and six respectively. When the address 010_10 is fetched, index two would be examined first, followed by index six. A miss occurs, the data from index two is moved to index six, and 010_10 is loaded into index two. The address 000_10 at index six would now be found using the second hashing function, and the corresponding rehash bit is set. Note that the more recently referenced block 001_10 was discarded. Now, address 001_10 is referenced. Index six is examined first; that block does not contain 001_10, but the rehash bit has been set, indicating that line two *can not* contain 001_10. Index two is not examined, and the miss is issued one cycle earlier than in the case of the HR-Cache.

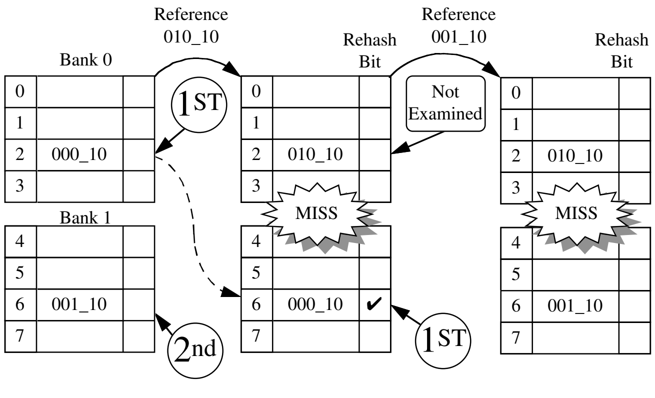
In the HR-Cache, the replacement policy always replaces the contents of the second block that is examined. However, as in our example, that block may have been more recently referenced than the first block. Now consider the CA-Cache: if a miss occurs and the rehash bit for the first block is set, we know the first block contains older data than the second block. If the data in the first block was more recent, it would have been swapped to the other block by some previous reference, setting the rehash bit. If the rehash bit is set, the CA-Cache replacement policy replaces the value in the first block. If the rehash bit is not set, the contents of the first block are transferred to the second block, and the fetched data is stored in the first block. Our detailed example illustrates a situation where the CA-Cache does not implement a true LRU replacement algorithm; however, the CA-Cache does have a miss rate close to that of true LRU replacement.

2.2 Dynamically Ordered Cache Probes

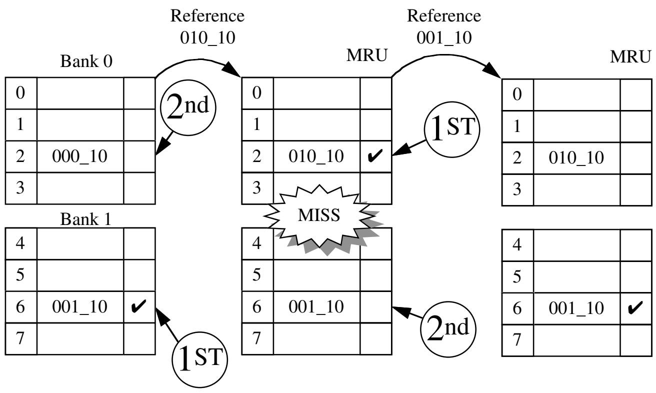
In contrast with the HR-Cache and CA-cache schemes, other researchers have developed schemes that use a dynamic probe ordering and do not rely on special cache allocation or re-allocation policies. Chang *et al* [4] proposed a novel organization for a multi-chip 32-way set-associative cache. Their study, and an earlier



(a) Hash-Rehash Cache



(b) Column Associative Cache



(c) MRU Cache

Figure 1: Cache Organizations

study by So and Rechtschaffen [11], found that most requests reference the most recently used blocks. Each set of cache blocks had associated "most recently used" (MRU) information. When accessing a cache set, the cache provided the block selected by the MRU information. Concurrently, the tags from the different chips were gathered and compared; if the wrong cache item was used, it was detected and the proper item was provided on the next cycle. This organization allowed most references (85%-95%) to complete in a single cycle, and reduced the cycle time of their multi-chip implementation by 30-35%.

Kessler *et al* [8] proposed a similar organization. Rather than swap the cache locations like the HR-Cache and CA-Cache, each pair of cache blocks uses an "MRU bit" to indicate the most recently used block. When searching for data, the block indicated by the MRU bit is probed first. If the data is not found, the second block is probed; if the data is found, the MRU bit is inverted, indicating the second block is more recently used than the first block. If the data is not found, the least recently used block, indicated by the MRU bit, is replaced. The MRU bit is used to implement an LRU replacement policy, so the MRU-Cache has the same miss rate as a two-way set-associative cache. The implementation of MRU two-way associative caches is shown in Figure 1(c). In that diagram, we show the MRU bits as check-marks to the side of each cache block. In practice, pairs of cache lines, such as lines two and six, share a single MRU bit. When 010_10 is referenced, block six, the most recently referenced block, is examined first. Since the data is not found, block two is examined. The missing data is fetched from memory and placed in block two. The MRU bit is set to indicate that block two is now more recent than block six. On the next reference, block two is examined first, followed by block six. The MRU bit is updated to indicate that block six was more recently used.

The design in [8] focused on large, secondary caches, and the lower cycle time seen by [4] did not apply. In the design by Kessler *et al*, the MRU bit must be fetched prior to accessing the cache contents to determine what cache line should be examined first, lengthening the cache access cycle, even if pipelined. It was felt this cache organization was appropriate for large secondary caches, because searches would be infrequent and the additional overhead for fetching the MRU bit could be speculatively overlapped with the first level access.

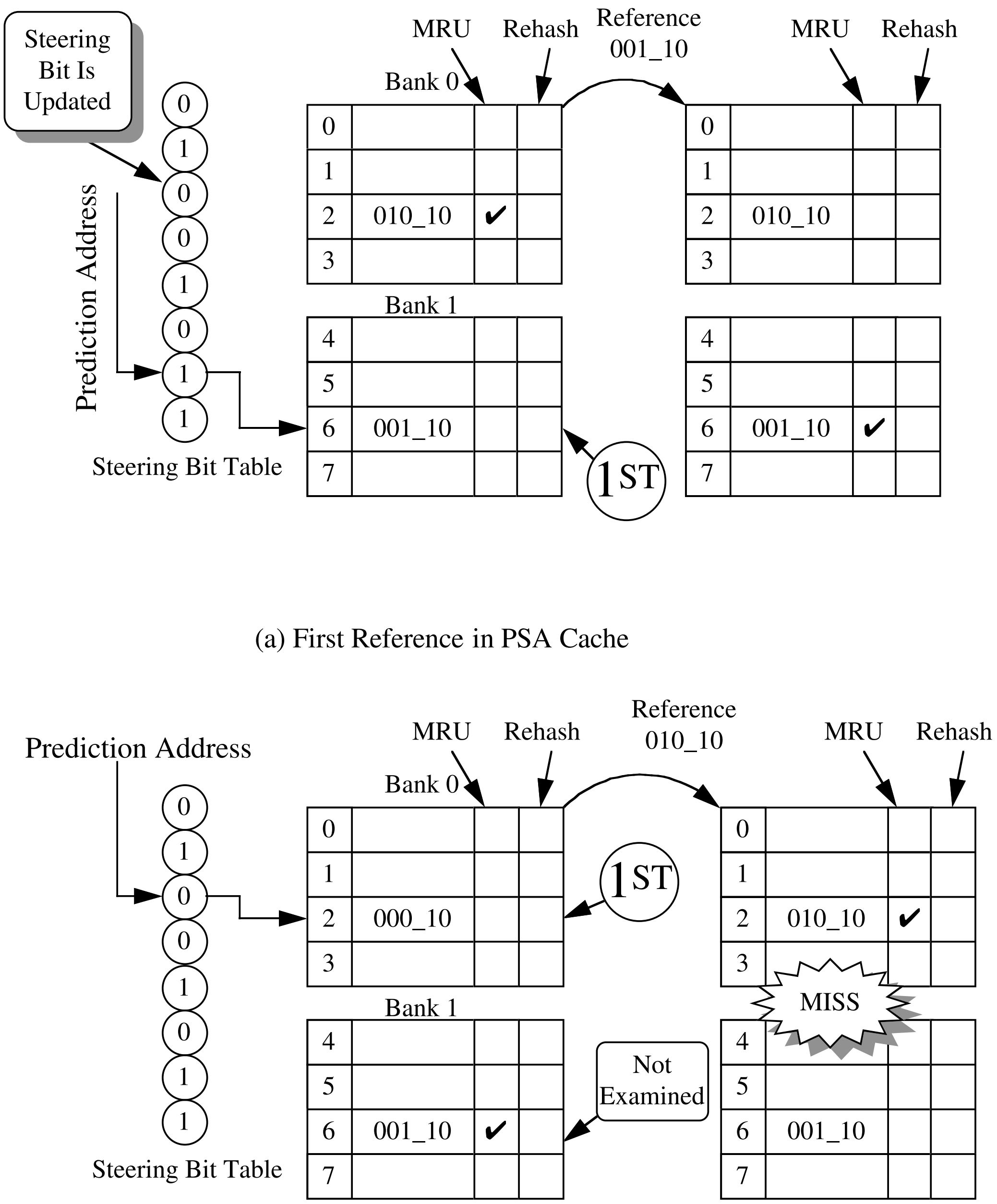
3 The Predictive Sequential Associative Cache

Both the HR-Cache and the CA-Cache require that entire cache lines be exchanged and have a worse miss rate than a cache with an LRU replacement strategy. In most first-level cache designs, the MRU-Cache requires a slightly longer cycle time to access the MRU prediction information. Furthermore, both the HR-Cache and MRU-Cache implementations suffer from excessive searching in certain situations. For example, assume the address references used in the previous examples have completed, and the processor continues requesting the alternating addresses 010_10, 001_10, . . . , 010_10, 001_10. In the HR-Cache, address 000_10 will continue to be moved between indices two and six, and each reference will be a miss; in the CA-Cache, these references do not result in further misses. In the MRU-Cache, each reference examines both indices two and six in the cache, because the pairs of blocks in the MRU-Cache share an MRU bit. The MRU information "flip-flops" on each reference, insuring that the next access requires two cycles.

These problems are addressed by our proposed cache design, the Predictive Sequential Associative Cache (PSA-Cache), shown in Figure 1(d). We separate the mechanism used to select probe order from the mechanism used to guide replacement. Each pair of cache blocks uses an MRU entry to implement LRU replacement. The PSA-Cache has the same miss rate as the MRU-cache and all other LRU-replacement caches. We use another table, the *steering bit table* (SBT), shown on the left side of Figure 1(d) to guide data access. When fetching a cache line entry, the effective address is used to index into the actual cache. Likewise, a *prediction index* is used to select a particular steering bit. As Kessler *et al* [8] indicated, the steering bits need to be accessed prior to the cache access. If we use the effective address to select a steering bit, this may lengthen the cache access time – arguably, if the effective address were available earlier, cache accesses would be initiated at an earlier pipeline stage. However, we do not need to use the effective fetch address to select a steering bit. We examined a number of sources for prediction indices, and present several very accurate sources that can be provided by earlier pipeline stages, insuring the steering bit is available when the cache is accessed.

Separating the replacement mechanism from the prediction mechanism offers immediate benefits, even for the MRU-Cache design proposed by Kessler *et al*. Consider an 8KByte cache split into two banks with 128 pairs of 32 byte lines. The MRU-Cache would use a 128-bit table to indicate the most recently used block in each pair. The PSA-Cache also uses a 128-bit table to implement an LRU replacement policy; however, a much larger table can be used to determine the block that should be probed first when searching for an address. Each entry "steers" references to the appropriate cache block. If a 256-entry SBT was used, the "flip-flop" example would encounter no penalty in the PSA-cache if different steering bit entries are used. In certain configurations, it is also useful to use a rehash bit in the PSA-Cache. As in the CA-Cache, we use this bit to avoid examining another line when that line can not possibly contain the requested address, but we do not use the bit to guide the replacement policy, since the MRU bit provides more accurate information.

Figure 2 shows the operation of the PSA-Cache, indicating both the MRU and rehash bits for each block. The rehash bit for 000_10 is clear because the 000_10 would be found on the first probe; likewise the rehash bit for line six is clear because 001_10 would also be found on the first probe. Figure 2(a) shows the reference to address 010_10. Prior to the access, a prediction source was mapped to the third entry in the steering bit table. That entry indicates the first block of the set, i.e., index two, should be probed first. Index two is examined first, and the rehash bit for line six is read concurrently. Index two does not contain 010_10. The rehash bit indicates the contents of block six is not a rehashed entry, and there is no point in examining index six. The referenced address is not in the cache, and is fetched from memory. The MRU bit indicates that the block at index six was more recently used than that at index two, so the contents of index two are replaced with 010_10. As the block is replaced, the steering bit used to locate 010_10 is trained, indicating



(a) First Reference in PSA Cache

(b) Second Reference in PSA Cache

Figure 2: Diagram of the PSA Cache

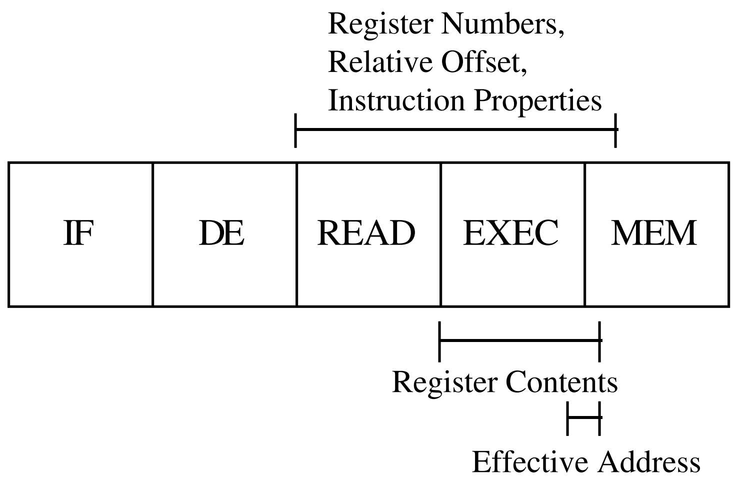


Figure 3: Pipeline Stages Showing When Prediction Sources Are Available

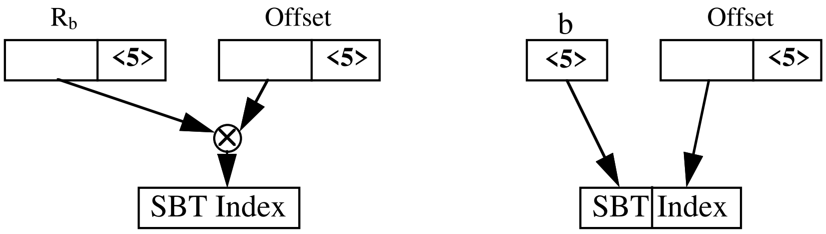
that bank zero of the set should be probed first on the next reference to 010_10. The prediction source selects the seventh steering bit when 001_10 is referenced. The requested address is found at index six, and the MRU bit is changed to indicate that index six is more recently used than index two. If the processor continues requesting the alternating addresses 010_10, 001_10, . . . , 010_10, 001_10, each reference will be found in the first probe.

In summary, we use three data structures to implement three cache mechanisms. The Steering Bit Table determines which block in a set should be probed first, increasing the number of references found during the first probe. The rehash bits reduce the number of probes, allowing misses to be started earlier or simply reducing the time the cache is busy, which is important for architectures that issue multiple loads per cycle. The MRU bits provide a true LRU replacement policy, improving the overall miss rate.

3.1 Prediction Sources

To illustrate some of the specific prediction sources available, Figure 3 shows a simple pipeline and the information available at each stage. We assume a load-store architecture with register-relative addressing – all memory references are of the form $M[R_b + \text{Offset}]$. At instruction fetch (4 cycles before the memory access), we know the instruction address. Following decode (3 cycles before the memory access), we know the register number (b) and the address offset (Offset). After the register file has been read (1 cycle prior to the memory access), we know the contents of R_b, and after the execution stage (right at memory access time), we know the effective address, $R_b + \text{Offset}$. In addition we can use the same information from prior instructions. We examined the following prediction sources:

1. *Effective Address*. The effective address was the most accurate prediction source; however, there may not be enough time in some designs to compute the effective address and index the steering bits before



(a) SBT Index for XOR-5-5 (b) SBT Index for RegNum

Figure 4: Combining Prediction Sources for Steering Bit Table Index

the cache access completes. When using the effective address, the PSA-Cache is a simple extension to the MRU-Cache with improved performance from a larger steering bit table.

2. *Register Contents and Offset.* Computing the effective address involves a full add. Functions without carry propagation take less time in some designs, thus making the results available in time to index the steering bit table before the cache access completes. We used the exclusive-or of the contents and offset to form a prediction address.

3. *Register Number and Offset.* We can combine the register number and the offset several cycles before the cache access. In general, this provides good performance with small SBTs, but the performance improvements dropped off for larger tables. There were three reasons. We were using the *register number* and not the register contents. Register assignments, particularly at procedure calls, were not reflected in the prediction information. Also, our target architecture has 32 integer registers, and most values for Offset were less than 96; some combinations of the register number and offset did not spread references enough to make use of the entire SBT. Lastly, some registers were used more than others; in most programs, $\approx$ 40% of all references were relative to the stack pointer.

   We included the *stack depth* to reduce interference between register usage in different procedures. We further improved this by including the address of the current procedure. Some of this information is already retained by many machines to implement a return-address stack [7], a branch prediction mechanism used to predict procedure return addresses. We also tried separate steering bit tables for certain registers.

4. *Instruction and Previous References.* We also used the address of the instruction issuing the reference and variants of the previous cache reference. These prediction sources were less effective than the others, and are not discussed further.

We examine four configurations of the PSA-Cache in more detail: the “Eff”, “XOR-5-5”, “RegNum” and “Proc” caches. We simulated an 8KByte cache with 32 byte lines. Each cache reference is of the form $M[R_b + \text{offset}]$, and the cache contains 256 32-byte lines. The lines are divided into 128 sets. All the cache models use $(R_b + \text{offset}) >> 5$, or $R_b + \text{offset}$ shifted right five bits, to index into the MRU table. The SBT entry determines the first bank to be examined. Each cache uses a 1024-entry SBT table.

The “Eff” configuration uses $(R_b + \text{offset}) \gg 5$ to index both the cache and the SBT table. This configuration illustrates the benefits of changing the MRU-Cache to use a larger steering bit table. Figure 4 illustrates how the indices for XOR-5-5 and RegNum are formed. The "XOR-5-5" configuration uses $(R_b \bigoplus \text{offset}) \gg 5)$, where $\bigoplus$ is a bit-wise exclusive or, to index the SBT. In some designs, there may be enough time to compute the exclusive-or and index the small SBT table before the effective address is computed, while the arithmetic sum $(R_b + \text{offset}) \gg 5$ would take longer to complete. The "RegNum" model forms the prediction address by concatenating the register number and the lower five bits of the offset ($(b \ll 5)|((\text{offset} \gg 5)\&\texttt{0x1F})$). The "Proc" configuration extends "RegNum" using an exclusive or of the destination address from the previous procedure call. Steering-bits resemble a single-bit branch prediction table; each entry contains a single bit and has no associated tag. Thus, for an 8KByte cache with 32 byte lines, a 1024-entry Steering Bit Table (SBT) represents $\approx$ 1% overhead. The actual overhead depends on the mechanism and design of the SBT.

4 Experimental Design and Performance Metrics

We compared the accuracy of the different prediction sources and the performance of the different cache organizations using trace-driven simulation. We collected information from 26 C and Fortran programs. We instrumented the programs from the SPEC92 benchmark suite and other programs, including many from the Perfect Club [3]. We used ATOM [12] to instrument the programs. Due to the structure of ATOM, we did not need to record traces and traced the full execution of each program. The programs were compiled on a DEC 3000-400 using the Alpha AXP-21064 processor and either the DEC C or FORTRAN compilers. Most programs were compiled using the standard OSF/1 V1.2 operating system. All programs were compiled with standard optimization (`-O`).

In this paper, we are primarily concerned with first-level data cache references, because data references are difficult to predict, and first level caches must be both fast and have low miss rates. Furthermore, instruction cache misses can be reduced using a number of software techniques [9, 10] and instruction references are usually very predictable. Thus, even the "Eff" technique described below can be used with instruction caches. We examined an 8 KByte cache with 32-byte cache lines. We assume the cache uses a *write-around* or *no-store-allocate* write policy, since earlier work by Jouppi [6] found this to be more effective than a *fetch-on-write* or *store-allocate* policy. The study by Jouppi found an overall lower miss rate using write-around. Our simulations show a slightly higher miss rate, particularly for writes.

5 Trace-Driven Performance Comparison

The cache miss rate is normally used to compare the performance of different cache organizations. However, we have seen that the access time for direct-mapped caches and traditional set-associative caches differ by as much as 50%, and this increased access time is not reflected in the miss rate. Furthermore, the traditional two-way set-associative cache, the MRU-Cache, and the PSA-Cache all use an LRU replacement algorithm,

Program	# of Instructions	% of Loads	% of Stores
APS	1,490,454,770	24.70	11.80
CSS	379,319,722	31.76	9.07
LGS	955,807,677	19.95	10.51
LWS	14,183,394,882	22.96	9.47
NAS	3,603,798,937	22.68	9.07
OCS	5,187,329,629	21.67	21.81
TFS	1,694,450,064	26.55	11.38
TIS	1,722,430,820	26.91	13.11
WSS	5,422,412,141	22.66	8.89
alvinn	5,240,969,586	26.95	9.30
dodoc	1,149,864,756	29.32	7.02
ear	17,005,801,014	22.09	12.67
fpppp	4,333,190,877	35.31	12.64
hydro2d	5,682,546,752	24.22	8.30
mdljsp2	3,343,833,266	22.53	6.52
nasa7	6,128,388,651	28.86	11.12
ora	6,036,097,925	22.26	9.75
spice	16,148,172,565	32.61	4.08
su2cor	4,776,762,363	22.39	10.21
wave5	3,554,909,341	21.33	13.39
compress	92,629,658	26.38	9.47
eqntott	1,810,540,418	12.77	1.29
espresso	513,008,174	21.57	5.08
gcc	143,737,915	23.89	11.74
li	1,355,059,387	28.09	14.65
sc	1,450,134,411	13.45	5.75

Table 1: Measured attributes of traced programs showing the number of instructions executed during execution and the percentage of loads and stores.

and have identical miss rates. However, the PSA-Cache and MRU-Cache may probe the cache several times to achieve that same miss rate, and a "probe rate" may be a more appropriate metric. Furthermore, when comparing the MRU-Cache and PSA-Cache to the HR-Cache and CA-Cache, we must also include the differences in miss rates. Finally, when comparing any of these methods to a two-way associative cache, we should include the difference in cycle time between an associative cache and the direct mapped caches used to implement the sequential associative caches.

We decided to compare the techniques using a timing model that separates the latency encountered by the pipeline and the time the cache is busy. Agarwal [2] used a simple timing model to demonstrate the performance of the CA-Cache. His model provides an average access time and can be used to compare all cache organizations that have the same cycle time. However, Agarwal's timing model did not distinguish between loads and stores. Conceptually, a processor pipeline must wait until a load is resolved, but need not wait for a store to finish – in practice, several loads and stores may be waiting to be resolved. Even if the processor is able to continue to issue loads after a miss, the pending miss may interfere with the loads that hit in the cache.

5.1 Performance Metrics

We define the *cache access latency* to be the average time the processor must wait for a memory reference to be resolved. Similarly, the *average cache occupancy* is the time the cache is busy for each reference. In general, a smaller access latency and smaller occupancy is preferred. If the latency is high, the processor must stall, waiting for data. If the occupancy is high, there is a greater chance outstanding references will conflict with newly issued references. As we show later, most of the cache designs have the same access latency, and are differentiated by their cache occupancy. It is difficult to precisely quantify the performance resulting from a particular access latency and occupancy, because system performance depends on instruction scheduling, the number of out-standing references, the depth of write-buffers and a number of other features determined by a particular system. However, latency and occupancy, like miss rates, can be used to narrow the design space prior to system-level simulation.

Table 2 defines certain parameters used in our timing model, and Table 3 shows how we calculate access latency and occupancy. We record different hit rates for loads and stores because loads and stores are treated differently, although that distinction is not made explicit in the timing equations to simplify the notation. The sequential associative caches further divide the hit rate H into hits that are detected on the first cache probe, H_f, and those detected on the second probe H_s. The CA-Cache and PSA-Cache use the rehash bit to avoid a second cache probe for some cache misses. In the sequential caches, the miss rate M is divided into M_f, denoting the misses detected on the first cache probe, and M_s denoting the misses detected on the second cache probe. The HR-Cache and MRU-Cache always probe the cache twice on misses, and M_f is always zero for these caches.

The latency for a cache miss, or miss penalty, is T_M cycles. This includes the time to request the data

T_P	Time to probe the cache following the first probe, in cycles. In some designs, this may be larger than one cycle, but we assume it is one cycle.
T_M	Penalty for cache misses, in cycles. This includes the time to initiate the cache miss and receive the data.
T_R	Time to refill a cache line, in cycles. This is the time the cache is busy when a cache line is refilled. We assume a 32-byte cache line can be refilled in two cycles.
T_{NS}	Extra time needed if misses can not be squashed. The "Conservative" timing model assumes $T_{NS} = T_P$, while the "Optimistic" timing model assumes $T_{NS} = 0$.
T_S	Time needed to swap cache lines in the HR-Cache & CA-Cache.

Table 2: Definition of Terms Used in Timing Equations

		Cache Access Latency	Cache Occupancy Time
Direct & 2-Way	Load	$H + (1 + T_M)M$	$H + (1 + T_R)M$
	Store	0	H
HR-Cache & CA-Cache	Load	$H_f + (1 + T_P)H_s + (1 + T_M)M_f + (1 + T_{NS} + T_M)M_s$	$H_f + (1 + T_P + T_S)H_s + (1 + T_R)M_f + (1 + T_P + T_S + T_R)M_s$
	Store	0	$H_f + (1 + T_P + T_S)H_s + M_f + (1 + T_P)M_s$
MRU & PSA	Load	$H_f + (1 + T_P)H_s + (1 + T_M)M_f + (1 + T_{NS} + T_M)M_s$	$H_f + (1 + T_P)H_s + (1 + T_R)M_f + (1 + T_P + T_R)M_s$
	Store	0	$H_f + (1 + T_P)H_s + M_f + (1 + T_P)M_s$

Table 3: Timing Equations Used To Compare Performance. In the HR-Cache and MRU-Cache, all misses take two cycles, meaning that $M_f = 0$ and $M_s = M$. The raw cache access time is a single cycle. We assume $T_P = 1$ and $T_S = 4T_R - 2$.

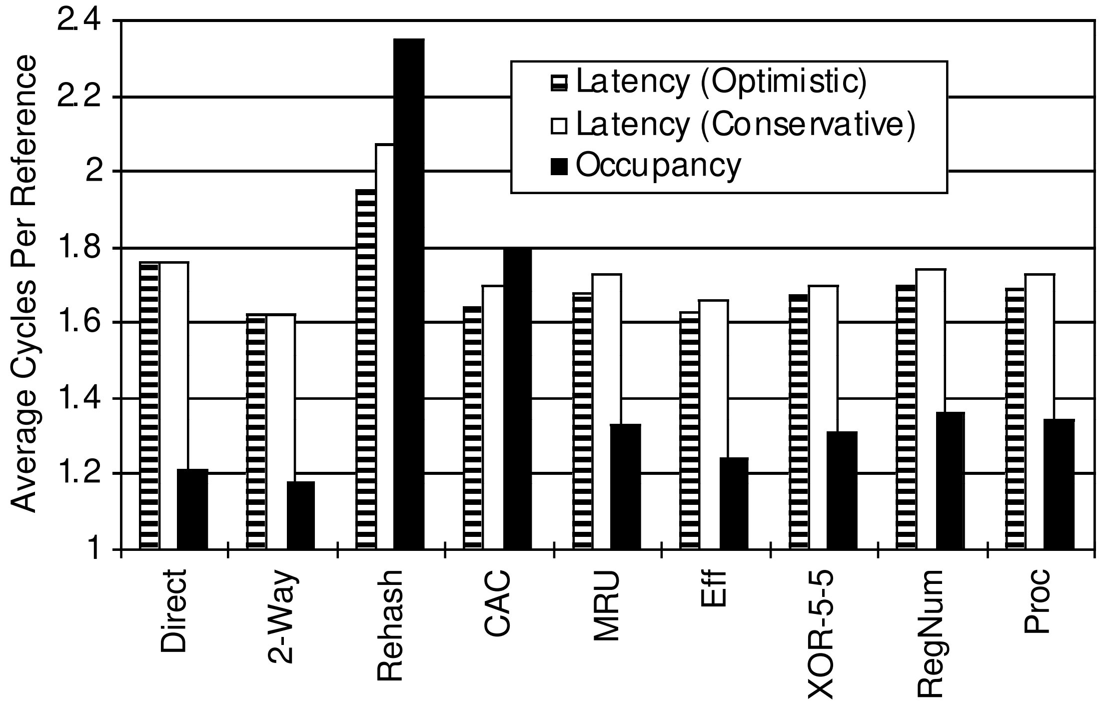


Figure 5: Latency and Occupancy for Conservative and Optimistic Configurations. The occupancy is the same for both the Conservative and Optimistic configurations. Parameters are $T_M = 10, T_R = 2, T_S = 4T_R - 2, T_P = 1$. Cache size is 8KBytes, with 32 byte lines. The values shown are the arithmetic means averaged over all programs.

from lower levels of the memory hierarchy, move it on-chip and load it into the cache. In some cases, a miss can be initiated speculatively and "squashed" in a later cycle. This reduces the latency for a miss, because the miss can be initiated a cycle earlier. Our "Conservative" cache timing model does not initiate misses speculatively, while our "Optimistic" timing model assumes a cache miss can be speculatively initiated and squashed one cycle later. We use the term T_{NS} ("non-squashing") to reflect the additional time spent servicing misses if the miss can not be speculatively initiated.

While the latency determines how long the processor must wait for references to be resolved, the occupancy determines how long the cache is busy. Although the miss penalty is T_M cycles, the cache is only busy when the cache lines are being reloaded, or T_R cycles. The HR-Cache and CA-Cache exchange cache lines to improve the average access latency and we assume it takes T_S cycles to swap two cache lines. Agarwal's timing model [2] combines the notion of latency and occupancy, and argued that T_S should be a single cycle. We feel a larger value is more reasonable, particularly for large cache lines, such as 32 or 128 bytes. There would be increased wiring density needed to exchange two complete cache lines in a single cycle, possibly increasing the cycle time. If we can reload half a cache line in a single cycle (i.e., $T_R = 2$), it could be argued that we could exchange a cache line in $4T_R - 2$ cycles, because two half-lines have already been read by the time we determine the lines must be swapped.

Table 3 shows the timing model. Stores do not stall the processor, and have a latency of zero. Consider the load latency for the CA-Cache. If the first probe results in a hit, a cycle was spent. If the second probe is a hit, the pipeline was stalled two cycles. At this point, the cache lines must be swapped, but the pipeline does not wait for this to finish. However, the cache lines must be swapped before the next cache reference begins. If a reference misses in the first bank, and the rehash bit indicates that it can not hit in the second bank, the pipeline stalls for $1 + T_M$ cycles. If the second bank must be examined, the processor stalls for $1 + T_M$ cycles, and may stall for an additional T_{NS} cycles if the miss can not be started early. In the CA-Cache, the cache is occupied while the cache is probed, the cache lines are swapped or cache lines are refilled.

5.2 Performance Comparison

The graph in Figure 5 summarizes the latency and occupancy, while Tables 4, 5 and 6 provide more detailed information. In each configuration, the cache miss penalty is ten cycles ($T_M = 10$), it takes two issues to refill a line ($T_R = 2$) and six cycles to swap cache lines ($T_S = 4T_R - 2$). Each cache is 8 KBytes with 32 byte lines. The latency is given for both the conservative timing model, where cache misses can not be speculatively initiated, and the optimistic model, where they can. The occupancy does not depend on the ability to speculatively initiate cache misses, and is the same for both timing models. Latency and occupancy are measured in average cycles per reference. Although Tables 4 and 5 show the latency and occupancy can be less than one, the averages shown in Figure 5 are not, and the vertical axis is bounded by one.

						PSA Cache			
Program	Direct	2-Way	Rehash	CAC	MRU	Eff	XOR-5-5	RegNum	Proc
APS	1.24	1.13	1.31	1.13	1.17	1.13	1.17	1.20	1.19
CSS	1.44	1.17	1.39	1.21	1.23	1.18	1.23	1.27	1.24
LGS	0.88	0.81	0.86	0.83	0.82	0.81	0.86	0.89	0.87
LWS	1.11	0.91	1.11	0.94	0.95	0.91	0.97	0.98	0.98
NAS	1.43	1.20	1.39	1.24	1.24	1.20	1.22	1.23	1.22
OCS	1.36	1.30	1.35	1.32	1.31	1.30	1.31	1.33	1.33
SDS	1.01	1.04	1.17	1.02	1.06	1.04	1.10	1.11	1.11
TFS	1.30	1.25	1.29	1.26	1.26	1.25	1.29	1.29	1.30
TIS	1.54	1.30	1.73	1.32	1.40	1.30	1.36	1.49	1.52
WSS	1.48	1.38	1.51	1.39	1.42	1.39	1.46	1.50	1.52
Perf Club Mean	1.28	1.15	1.31	1.17	1.19	1.15	1.20	1.23	1.23
Perf Club StdDev	0.22	0.18	0.23	0.18	0.19	0.18	0.18	0.20	0.21
alvinn	1.30	1.22	1.33	1.22	1.24	1.22	1.22	1.23	1.23
doduc	1.70	1.37	1.71	1.41	1.45	1.38	1.42	1.45	1.43
ear	0.88	0.79	1.07	0.80	0.83	0.79	0.80	0.81	0.81
fpppp	1.23	0.90	1.16	0.94	0.96	0.90	1.03	1.07	1.02
hydro2d	2.12	1.96	2.11	2.00	1.99	1.97	2.05	2.02	2.05
mdljsp2	1.14	1.01	1.27	1.03	1.06	1.02	1.03	1.06	1.06
nasa7	3.86	3.69	3.80	3.72	3.78	3.70	3.76	3.75	3.76
ora	0.96	0.70	0.79	0.72	0.72	0.70	0.74	0.75	0.72
spice	3.58	3.28	3.60	3.33	3.38	3.29	3.35	3.43	3.42
su2cor	4.18	4.14	4.34	4.13	4.18	4.14	4.19	4.22	4.20
swm256	2.52	3.03	5.29	2.92	3.32	3.03	3.08	3.14	3.14
tomcatv	3.58	3.83	5.92	3.95	4.12	3.83	3.89	3.94	3.98
wave5	1.30	1.15	1.57	1.15	1.22	1.15	1.17	1.26	1.25
SPECfp Mean	2.18	2.08	2.61	2.10	2.17	2.09	2.13	2.16	2.16
SPECfp StdDev	1.22	1.31	1.75	1.31	1.36	1.31	1.32	1.32	1.33
compress	2.22	1.86	1.98	1.92	1.90	1.86	1.86	1.91	1.89
eqntott	1.49	1.33	1.48	1.35	1.36	1.33	1.34	1.36	1.35
espresso	1.36	1.25	1.33	1.26	1.28	1.25	1.25	1.31	1.30
gcc	1.32	1.08	1.31	1.12	1.13	1.08	1.13	1.19	1.16
li	1.34	0.99	1.23	1.03	1.05	0.99	1.06	1.13	1.09
sc	2.16	1.99	2.23	2.02	2.03	1.99	2.00	2.06	2.03
SPECint Mean	1.65	1.41	1.59	1.45	1.46	1.42	1.44	1.49	1.47
SPECint StdDev	0.43	0.41	0.41	0.42	0.41	0.41	0.39	0.39	0.39
Overall Mean	1.76	1.62	1.95	1.64	1.68	1.63	1.67	1.70	1.69
Overall StdDev	0.92	0.98	1.32	0.98	1.02	0.98	0.98	0.99	0.99

Table 4: Cache Access Latency for Optimistic Timing Model with $T_M = 10$ Cycles, $T_R = 2$ Cycles, $T_S = 4T_R - 2$ Cycles.

						PSA Cache			
Program	Direct	2-Way	Rehash	CAC	MRU	Eff	XOR-5-5	RegNum	Proc
APS	1.24	1.13	1.37	1.17	1.20	1.16	1.20	1.23	1.22
CSS	1.44	1.17	1.45	1.24	1.25	1.19	1.25	1.29	1.26
LGS	0.88	0.81	0.88	0.84	0.83	0.82	0.86	0.89	0.87
LWS	1.11	0.91	1.15	0.96	0.96	0.92	0.98	0.99	0.99
NAS	1.43	1.20	1.46	1.28	1.27	1.23	1.25	1.26	1.25
OCS	1.36	1.30	1.43	1.37	1.37	1.33	1.35	1.37	1.37
SDS	1.01	1.04	1.21	1.03	1.07	1.05	1.11	1.12	1.12
TFS	1.30	1.25	1.35	1.29	1.28	1.27	1.31	1.31	1.32
TIS	1.54	1.30	1.83	1.36	1.42	1.32	1.38	1.51	1.54
WSS	1.48	1.38	1.59	1.43	1.45	1.41	1.49	1.53	1.55
Perf Club Mean	1.28	1.15	1.37	1.19	1.21	1.17	1.22	1.25	1.25
Perf Club StdDev	0.22	0.18	0.26	0.19	0.20	0.19	0.19	0.20	0.22
alvinn	1.30	1.22	1.38	1.24	1.25	1.23	1.23	1.24	1.24
doduc	1.70	1.37	1.80	1.44	1.47	1.40	1.44	1.48	1.45
ear	0.88	0.79	1.11	0.81	0.83	0.79	0.80	0.81	0.81
fpppp	1.23	0.90	1.20	0.95	0.97	0.91	1.04	1.07	1.03
hydro2d	2.12	1.96	2.24	2.07	2.04	2.00	2.09	2.07	2.11
mdljsp2	1.14	1.01	1.32	1.05	1.08	1.02	1.04	1.07	1.07
nasa7	3.86	3.69	4.11	3.91	3.90	3.81	3.88	3.89	3.90
ora	0.96	0.70	0.80	0.72	0.72	0.70	0.74	0.75	0.72
spice	3.58	3.28	3.86	3.48	3.49	3.39	3.46	3.54	3.54
su2cor	4.18	4.14	4.71	4.31	4.36	4.24	4.30	4.37	4.35
swm256	2.52	3.03	5.75	3.03	3.40	3.10	3.15	3.22	3.22
tomcatv	3.58	3.83	6.43	4.12	4.25	3.89	3.99	4.06	4.11
wave5	1.30	1.15	1.67	1.18	1.24	1.17	1.19	1.28	1.27
SPECfp Mean	2.18	2.08	2.80	2.18	2.23	2.13	2.18	2.22	2.22
SPECfp StdDev	1.22	1.31	1.93	1.38	1.42	1.35	1.36	1.38	1.39
compress	2.22	1.86	2.10	2.00	1.96	1.91	1.91	1.96	1.94
eqntott	1.49	1.33	1.53	1.38	1.38	1.35	1.36	1.38	1.37
espresso	1.36	1.25	1.38	1.29	1.30	1.27	1.27	1.33	1.32
gcc	1.32	1.08	1.37	1.15	1.15	1.10	1.15	1.21	1.18
li	1.34	0.99	1.28	1.05	1.06	1.01	1.07	1.15	1.11
sc	2.16	1.99	2.38	2.10	2.09	2.04	2.05	2.11	2.09
SPECint Mean	1.65	1.41	1.67	1.49	1.49	1.44	1.47	1.53	1.50
SPECint StdDev	0.43	0.41	0.45	0.44	0.43	0.43	0.41	0.41	0.41
Overall Mean	1.76	1.62	2.07	1.70	1.73	1.66	1.70	1.74	1.73
Overall StdDev	0.92	0.98	1.45	1.04	1.06	1.01	1.02	1.03	1.04

Table 5: Cache Access Latency for Conservative Timing Model with $T_M = 10$ Cycles, $T_R = 2$ Cycles, $T_S = 4T_R - 2$ Cycles.

						PSA Cache			
Program	Direct	2-Way	Rehash	CAC	MRU	Eff	XOR-5-5	RegNum	Proc
APS	1.11	1.09	1.78	1.54	1.18	1.14	1.19	1.25	1.23
CSS	1.13	1.08	1.80	1.52	1.17	1.11	1.18	1.23	1.20
LGS	1.05	1.03	1.33	1.22	1.09	1.06	1.14	1.18	1.14
LWS	1.08	1.04	1.58	1.36	1.11	1.06	1.17	1.18	1.17
NAS	1.14	1.10	1.82	1.55	1.19	1.14	1.19	1.20	1.17
OCS	1.17	1.16	2.00	1.71	1.31	1.22	1.27	1.31	1.32
SDS	1.06	1.06	1.44	1.22	1.11	1.08	1.17	1.19	1.20
TFS	1.12	1.11	1.70	1.48	1.20	1.18	1.25	1.26	1.27
TIS	1.17	1.13	2.20	1.66	1.25	1.15	1.24	1.39	1.41
WSS	1.15	1.13	1.93	1.61	1.24	1.18	1.30	1.35	1.37
Perf Club Mean	1.12	1.09	1.76	1.49	1.19	1.13	1.21	1.25	1.25
Perf Club StdDev	0.05	0.04	0.26	0.17	0.07	0.05	0.05	0.07	0.09
alvinn	1.11	1.09	1.62	1.37	1.13	1.11	1.11	1.13	1.12
doduc	1.18	1.11	2.19	1.75	1.26	1.17	1.23	1.27	1.23
ear	1.05	1.03	1.52	1.22	1.10	1.04	1.06	1.08	1.07
fpppp	1.10	1.03	1.73	1.46	1.15	1.07	1.23	1.28	1.23
hydro2d	1.28	1.24	2.50	1.97	1.38	1.31	1.45	1.43	1.46
mdljsp2	1.07	1.05	1.56	1.27	1.12	1.06	1.08	1.11	1.10
nasa7	1.63	1.59	4.24	3.37	1.86	1.75	1.84	1.85	1.85
ora	1.05	1.00	1.28	1.18	1.03	1.00	1.06	1.08	1.02
spice	1.54	1.48	3.80	2.99	1.70	1.60	1.67	1.76	1.75
su2cor	1.70	1.69	4.70	3.42	2.05	1.91	1.98	2.07	2.04
swm256	1.36	1.46	5.41	2.55	1.94	1.56	1.65	1.73	1.73
tomcatv	1.56	1.61	5.92	3.14	2.16	1.71	1.84	1.91	1.96
wave5	1.14	1.11	2.23	1.65	1.30	1.20	1.24	1.35	1.34
SPECfp Mean	1.29	1.27	2.98	2.10	1.48	1.35	1.42	1.46	1.45
SPECfp StdDev	0.24	0.26	1.62	0.87	0.41	0.32	0.34	0.35	0.36
compress	1.30	1.22	2.38	2.05	1.34	1.28	1.28	1.35	1.32
eqntott	1.12	1.08	1.63	1.41	1.14	1.11	1.13	1.15	1.13
espresso	1.11	1.09	1.60	1.41	1.16	1.12	1.13	1.20	1.18
gcc	1.13	1.08	1.87	1.55	1.20	1.13	1.21	1.29	1.24
li	1.14	1.07	1.83	1.53	1.17	1.10	1.20	1.31	1.26
sc	1.29	1.26	2.68	2.09	1.44	1.38	1.40	1.49	1.45
SPECint Mean	1.18	1.13	2.00	1.67	1.24	1.19	1.22	1.30	1.27
SPECint StdDev	0.09	0.08	0.44	0.31	0.12	0.12	0.10	0.12	0.11
Overall Mean	1.21	1.18	2.35	1.80	1.33	1.24	1.31	1.36	1.34
Overall StdDev	0.18	0.19	1.23	0.66	0.31	0.24	0.25	0.26	0.27

Table 6: Average Cache Occupancy with $T_M = 10$ Cycles, $T_R = 2$ Cycles, $T_S = 4T_R - 2$ Cycles. The Cache Occupancy is the same for the the Conservative ($T_{NS} = 1$) and Optimistic ($T_{NS} = 0$) models.

In general, the latency for the associative caches are $\approx 5 - 10\%$ smaller than that of the direct mapped cache; exact values can be found in the tables. It is important to understand that this paper is not comparing the effectiveness of direct *vs.* associative caches; we assume that associative caches are desired, and an efficient implementation technique is needed. In our performance comparison, we examined caches with a small miss penalty, $T_M = 10$, because we feel the PSA-Cache is appropriate for first level caches. As the miss penalty increases, all two-way associative caches further reduce the latency, due to the reduce miss rate. The access time for a two-way associative cache depends on the cache size and a number of other factors. As mentioned, the access time for a two-way associative cache is 1.51, 1.46 and 1.40 times longer than the access time for a direct mapped cache for 8KB, 16KB and 32KB caches, respectively. It is incorrect to simply scale the cycles per memory reference show in Tables 4, 5 and 6 by these values, since the data cache access time typically limits the system cycle time and has a much broader impact on system performance. For example, assume we design a system using an 8KByte cache with a 3 nanosecond clock. Table 4 would imply that a two-way set associative cache would lower the average cycles per memory reference by 10-15%, by reducing the miss rate. However, such a cache would also be 1.51 times slower, with a 4.53ns cycle time. By comparison, the sequential cache configurations shown in Table 4 maintain the same 3ns cycle time, and reduce the average cycles per reference.

The latencies for most of the two-way associative caches are almost identical; this is not surprising, since the equations for latency in Table 3 are identical for the HR, CA, MRU and PSA caches. Any difference in the latency arises from different hit rates, and the fraction of references resolved on the first or second cycle. Only the HR-Cache has a notably higher miss rate. Table 4 demonstrates several points also seen in the remaining tables. First, for some programs (`swm256` and `tomcatv`), two-way set-associativity *increases* the miss rate. For the remaining programs, the ideal two-way set associative cache has the best performance, followed by the PSA-Cache using the "Eff" prediction address; however, this configuration simply extends the MRU-Cache to use a larger table of prediction bits, and it may not be possible to use the effective address to index a table of steering bits and access the data in a single cycle. However, the "Eff" column demonstrates how to improve the MRU-Cache design of Kessler *et al* for the domains considered in [8]. The next most effective configuration is "XOR-5-5," followed by the CA-Cache. The "XOR-5-5" configuration requires an exclusive-or of the contents of the register and offset before the SBT is accessed; this may not be possible in some designs. However, the "Proc" design provides almost equal performance with considerably more flexible timing constraints. The prediction sources for the "Proc" configuration are available immediately after the instruction is decoded.

In Table 6, the direct and traditional two-way cache have the lowest occupancy; this is understandable, because all cache operations either take one cycle (hit) or $1 + T_R$ cycles (miss). In the CA-Cache and PSA-Cache, rehash bits are used to avoid examining the second half of the cache in some situations. Agarwal [2] used the rehash bit to reduce the latency by initiating misses one cycle earlier. In our "Optimistic" timing model, the rehash bit has no effect on latency because misses are always initiated early, but rehash bits influence occupancy in all configurations. There is still a notable difference between the “Optimistic” and “Conservative” timing model, even when the rehash bits are used, indicating that the speculative miss initiation is useful even when the rehash bit can not avoid probing the cache a second time.

We feel that occupancy is an important metric, because it determines how quickly memory references, both loads and stores, can be issued without contention in the cache. Occupancy directly affects cache latency, but is highly dependent on machine and system architectures and instruction scheduling.

6 Conclusions

In this paper, we have primarily focused on the Predictive Sequential Associative Cache as a mechanism to implement two-way associative on-chip caches. We proposed two metrics, latency and occupancy, suitable for comparing associative cache designs. Variants of the PSA-Cache have better performance, in terms of latency and occupancy, than other proposed designs. We feel the PSA-Cache variants are easier to implement than designs that exchange cache lines, particularly for larger cache lines.

Our simultation study showed that all the techniques had comparable latency, with variants of the PSA-Cache having the lowest latency. The PSA-Cache also had the lowest occupancy. The “Eff” design has the best performance, but the “XOR-5-5” may be easier to implement.

There are a number of other design criteria not immediately evident from our performance metrics. First, the PSA-Cache may have fewer power requirements than other caches since a single bank is probed. Furthermore, since the PSA-Cache is divided into two banks that can be operated independently, it may be possible to support multiple references without dual-porting the banks. A similar argument can be made for the CA-Cache, but it would require extensive book-keeping to maintain correctness while blocks are exchanged. Lastly, the PSA-Cache mechanism may also be appropriate for larger, secondary caches directly controlled by the processor. The Steering Bit Table can be small, and implemented on the processor, while the MRU Table, cache tags and data can be implemented off-chip.

Acknowledgements

We would like to thank Alan Eustace and Amitabh Srivastava for developing ATOM. Brad Calder was supported by an ARPA Fellowship in High Performance Computing administered by the Institute for Advanced Computer Studies, University of Maryland This work was funded in part by NSF grant No. ASC-9217394, NSF grant No. CCR-9404669, ARPA contract ARMY DABT63-94-C-0029 and a software grant from Digital Equipment Corp.

References

[1] Anant Agarwal, John Hennesy, and Mark Horowitz. Cache performance of operating systems and multiprogramming. *ACM Transactions on Computer Systems*, 6:393–431, November 1988.

[2] Anant Agarwal and Steven D. Pudar. Column-associative caches: A technique for reducing the miss rate of direct mapped caches. In *20th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 179–190. IEEE, 1993.

[3] M. Berry. The Perfect Club Benchmarks: Effective performance evaluation of supercomputers. *The International Journal of Supercomputer Applications*, 3(3):5–40, Fall 1989.

[4] J. H. Chang, H. Chao, and K. So. Cache design of a sub-micron CMOS System/370. In *14th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 208–213. IEEE, June 1987.

[5] Mark Hill. A case for direct-mapped caches. *IEEE Computer*, 21(12):25–40, December 1988.

[6] Norm Jouppi. Cache write policies and performance. In *20th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 191–201. IEEE, May 1993.

[7] David R. Kaeli and Philip G. Emma. Branch history table prediction of moving target branches due to subroutine returns. In *18th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 34–42. ACM, May 1991.

[8] R. R. Kessler, Richard Jooss, Alvin Lebeck, and Mark D. Hill. Inexpensive implementations of set-associativity. In *16th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*. IEEE, May 1989.

[9] Scott McFarling. Program optimization for instruction caches. In *Proceedings of the 3rd Symposium on Architectural Support for Programming Languages and Operating Systems*, pages 183–191. ACM, 1988.

[10] Wen mei W. Hwu and Pohua P. Chang. Achieving high instruction cache performance with an optimizing compiler. In *16th Annual Annual International Symposium on Computer Architecture, SIGARCH Newsletter*, pages 242–251. ACM, ACM, 1989.

[11] Kimming So and Rudolph N. Rechtschaffen. Cache operations by MRU change. *IEEE Transactions on Computers*, 37(6):700–709, June 1988.

[12] Amitabh Srivastava and Alan Eustace. ATOM: A system for building customized program analysis tools. In *Proceedings of the SIGPLAN'94 Conference on Programming Language Design and Implementation*. ACM, 1994.

[13] Steven J. E. Wilton and Norman P. Jouppi. An enhanced access and cycle time model for on-chip caches. Report 93/5, DEC Western Research Lab, 1993.

Load Hit Rates for Different Applications

Direct								
Program	Loads				Stores			
	H_f	H_s	M_f	M_s	H_f	H_s	M_f	M_s
APS	0.62	0.00	0.06	0.00	0.27	0.00	0.05	0.00
CSS	0.71	0.00	0.07	0.00	0.18	0.00	0.04	0.00
LGS	0.63	0.00	0.02	0.00	0.28	0.00	0.07	0.00
LWS	0.67	0.00	0.04	0.00	0.26	0.00	0.03	0.00
NAS	0.64	0.00	0.07	0.00	0.24	0.00	0.05	0.00
OCS	0.41	0.00	0.09	0.00	0.32	0.00	0.18	0.00
SDS	0.70	0.00	0.03	0.00	0.26	0.00	0.02	0.00
TFS	0.64	0.00	0.06	0.00	0.17	0.00	0.13	0.00
TIS	0.59	0.00	0.09	0.00	0.32	0.00	0.01	0.00
WSS	0.64	0.00	0.08	0.00	0.21	0.00	0.08	0.00
alvinn	0.69	0.00	0.06	0.00	0.26	0.00	0.00	0.00
doduc	0.72	0.00	0.09	0.00	0.12	0.00	0.08	0.00
ear	0.61	0.00	0.02	0.00	0.33	0.00	0.03	0.00
fpppp	0.69	0.00	0.05	0.00	0.19	0.00	0.07	0.00
hydro2d	0.61	0.00	0.14	0.00	0.09	0.00	0.17	0.00
mdljsp2	0.74	0.00	0.04	0.00	0.22	0.00	0.01	0.00
nasa7	0.41	0.00	0.31	0.00	0.20	0.00	0.07	0.00
ora	0.67	0.00	0.03	0.00	0.30	0.00	0.01	0.00
spice	0.62	0.00	0.27	0.00	0.09	0.00	0.02	0.00
su2cor	0.34	0.00	0.35	0.00	0.08	0.00	0.23	0.00
swm256	0.56	0.00	0.18	0.00	0.09	0.00	0.17	0.00
tomcatv	0.49	0.00	0.28	0.00	0.11	0.00	0.12	0.00
wave5	0.55	0.00	0.07	0.00	0.21	0.00	0.18	0.00
compress	0.59	0.00	0.15	0.00	0.25	0.00	0.01	0.00
eqntott	0.85	0.00	0.06	0.00	0.08	0.00	0.01	0.00
espresso	0.75	0.00	0.06	0.00	0.16	0.00	0.04	0.00
gcc	0.61	0.00	0.06	0.00	0.25	0.00	0.08	0.00
li	0.59	0.00	0.07	0.00	0.30	0.00	0.04	0.00
sc	0.55	0.00	0.15	0.00	0.14	0.00	0.16	0.00

Rehash								
Program	Loads				Stores			
	H_f	H_s	M_f	M_s	H_f	H_s	M_f	M_s
APS	0.60	0.02	0.00	0.06	0.27	0.01	0.00	0.05
CSS	0.69	0.03	0.00	0.06	0.18	0.01	0.00	0.04
LGS	0.63	0.01	0.00	0.02	0.27	0.00	0.00	0.07
LWS	0.65	0.02	0.00	0.04	0.26	0.01	0.00	0.02
NAS	0.63	0.02	0.00	0.07	0.23	0.00	0.00	0.05
OCS	0.41	0.01	0.00	0.08	0.32	0.00	0.00	0.18
SDS	0.68	0.00	0.00	0.04	0.26	0.00	0.00	0.01
TFS	0.64	0.01	0.00	0.06	0.17	0.00	0.00	0.13
TIS	0.53	0.04	0.00	0.10	0.32	0.00	0.00	0.01
WSS	0.62	0.02	0.00	0.08	0.20	0.01	0.00	0.08
alvinn	0.67	0.01	0.00	0.06	0.26	0.00	0.00	0.00
doduc	0.68	0.04	0.00	0.09	0.12	0.01	0.00	0.06
ear	0.58	0.01	0.00	0.04	0.33	0.01	0.00	0.03
fpppp	0.66	0.03	0.00	0.04	0.18	0.01	0.00	0.07
hydro2d	0.59	0.02	0.00	0.13	0.09	0.00	0.00	0.17
mdljsp2	0.71	0.02	0.00	0.05	0.22	0.00	0.00	0.01
nasa7	0.38	0.04	0.00	0.30	0.20	0.02	0.00	0.06
ora	0.67	0.02	0.00	0.01	0.27	0.00	0.00	0.03
spice	0.57	0.05	0.00	0.27	0.09	0.00	0.00	0.02
su2cor	0.31	0.01	0.00	0.36	0.08	0.02	0.00	0.22
swm256	0.27	0.01	0.00	0.45	0.07	0.00	0.00	0.20
tomcatv	0.25	0.02	0.00	0.51	0.06	0.00	0.00	0.16
wave5	0.50	0.02	0.00	0.09	0.20	0.01	0.00	0.18
compress	0.58	0.04	0.00	0.12	0.25	0.00	0.00	0.01
eqntott	0.84	0.02	0.00	0.06	0.08	0.00	0.00	0.01
espresso	0.74	0.02	0.00	0.05	0.16	0.00	0.00	0.03
gcc	0.58	0.03	0.00	0.06	0.24	0.01	0.00	0.08
li	0.57	0.03	0.00	0.05	0.30	0.01	0.00	0.04
sc	0.53	0.02	0.00	0.15	0.14	0.00	0.00	0.16

CAC								
Program	Loads				Stores			
	H_f	H_s	M_f	M_s	H_f	H_s	M_f	M_s
APS	0.61	0.02	0.01	0.03	0.28	0.01	0.01	0.02
CSS	0.71	0.03	0.02	0.03	0.19	0.01	0.01	0.02
LGS	0.63	0.01	0.01	0.01	0.28	0.00	0.03	0.03
LWS	0.66	0.02	0.01	0.01	0.27	0.01	0.00	0.01
NAS	0.64	0.02	0.02	0.03	0.24	0.00	0.02	0.03
OCS	0.41	0.01	0.03	0.05	0.33	0.00	0.05	0.13
SDS	0.69	0.01	0.01	0.02	0.26	0.00	0.00	0.01
TFS	0.64	0.01	0.02	0.03	0.17	0.00	0.04	0.09
TIS	0.57	0.04	0.03	0.03	0.32	0.00	0.00	0.01
WSS	0.63	0.02	0.03	0.04	0.21	0.01	0.02	0.04
alvinn	0.68	0.01	0.02	0.02	0.26	0.00	0.00	0.00
doduc	0.71	0.04	0.02	0.03	0.12	0.01	0.02	0.03
ear	0.61	0.01	0.01	0.01	0.34	0.01	0.01	0.01
fpppp	0.69	0.03	0.01	0.01	0.20	0.01	0.01	0.04
hydro2d	0.60	0.02	0.05	0.07	0.09	0.00	0.07	0.10
mdljsp2	0.73	0.02	0.01	0.01	0.22	0.00	0.00	0.00
nasa7	0.38	0.04	0.11	0.18	0.21	0.02	0.01	0.04
ora	0.67	0.02	0.00	0.00	0.30	0.00	0.00	0.00
spice	0.59	0.06	0.08	0.15	0.09	0.00	0.01	0.01
su2cor	0.33	0.02	0.16	0.19	0.07	0.02	0.05	0.18
swm256	0.49	0.02	0.11	0.11	0.09	0.01	0.03	0.14
tomcatv	0.43	0.03	0.15	0.17	0.10	0.01	0.04	0.08
wave5	0.54	0.03	0.02	0.03	0.20	0.01	0.07	0.11
compress	0.58	0.04	0.04	0.08	0.25	0.00	0.00	0.01
eqntott	0.85	0.02	0.01	0.03	0.08	0.00	0.00	0.01
espresso	0.75	0.02	0.02	0.03	0.16	0.00	0.01	0.02
gcc	0.60	0.03	0.02	0.03	0.25	0.01	0.02	0.05
li	0.59	0.03	0.01	0.02	0.31	0.01	0.01	0.02
sc	0.55	0.02	0.05	0.08	0.14	0.01	0.03	0.12

MRU								
Program	Loads				Stores			
	H_f	H_s	M_f	M_s	H_f	H_s	M_f	M_s
APS	0.59	0.04	0.02	0.03	0.28	0.01	0.02	0.02
CSS	0.68	0.05	0.02	0.02	0.18	0.01	0.02	0.01
LGS	0.63	0.01	0.01	0.01	0.28	0.01	0.03	0.03
LWS	0.64	0.04	0.01	0.01	0.27	0.01	0.01	0.01
NAS	0.62	0.04	0.02	0.03	0.24	0.01	0.02	0.02
OCS	0.40	0.02	0.02	0.06	0.35	0.00	0.08	0.07
SDS	0.67	0.02	0.02	0.01	0.26	0.00	0.01	0.01
TFS	0.63	0.01	0.03	0.02	0.17	0.00	0.07	0.06
TIS	0.51	0.10	0.04	0.02	0.32	0.00	0.01	0.01
WSS	0.61	0.04	0.04	0.03	0.21	0.01	0.03	0.03
alvinn	0.67	0.03	0.04	0.01	0.26	0.00	0.00	0.00
doduc	0.67	0.08	0.03	0.02	0.12	0.02	0.03	0.03
ear	0.58	0.04	0.01	0.01	0.33	0.02	0.01	0.01
fpppp	0.66	0.06	0.01	0.01	0.19	0.02	0.02	0.03
hydro2d	0.59	0.03	0.08	0.05	0.09	0.00	0.11	0.06
mdljsp2	0.70	0.05	0.01	0.01	0.22	0.00	0.00	0.00
nasa7	0.33	0.09	0.17	0.12	0.21	0.03	0.02	0.02
ora	0.67	0.03	0.00	0.00	0.30	0.01	0.00	0.00
spice	0.55	0.10	0.12	0.12	0.09	0.00	0.01	0.01
su2cor	0.30	0.04	0.17	0.18	0.07	0.02	0.11	0.12
swm256	0.21	0.29	0.15	0.08	0.05	0.04	0.10	0.07
tomcatv	0.18	0.29	0.17	0.13	0.04	0.07	0.06	0.06
wave5	0.49	0.07	0.03	0.02	0.19	0.02	0.10	0.08
compress	0.58	0.05	0.06	0.06	0.25	0.00	0.00	0.00
eqntott	0.83	0.03	0.02	0.02	0.08	0.00	0.00	0.00
espresso	0.74	0.03	0.02	0.03	0.16	0.00	0.01	0.02
gcc	0.58	0.05	0.02	0.02	0.25	0.01	0.04	0.03
li	0.57	0.06	0.01	0.02	0.30	0.02	0.01	0.01
sc	0.53	0.04	0.07	0.06	0.14	0.01	0.08	0.07

Eff								
Program	Loads				Stores			
	H_f	H_s	M_f	M_s	H_f	H_s	M_f	M_s
APS	0.63	0.00	0.02	0.03	0.29	0.00	0.02	0.01
CSS	0.74	0.00	0.02	0.02	0.19	0.00	0.02	0.01
LGS	0.64	0.00	0.01	0.01	0.28	0.00	0.04	0.02
LWS	0.69	0.00	0.01	0.01	0.28	0.00	0.01	0.01
NAS	0.66	0.01	0.02	0.02	0.24	0.00	0.03	0.02
OCS	0.41	0.01	0.05	0.03	0.35	0.00	0.13	0.02
SDS	0.70	0.00	0.02	0.01	0.26	0.00	0.01	0.01
TFS	0.64	0.00	0.03	0.02	0.17	0.00	0.07	0.05
TIS	0.61	0.00	0.04	0.02	0.32	0.00	0.01	0.00
WSS	0.65	0.00	0.04	0.02	0.22	0.00	0.04	0.02
alvinn	0.69	0.00	0.04	0.01	0.26	0.00	0.00	0.00
doduc	0.74	0.01	0.04	0.02	0.13	0.01	0.04	0.02
ear	0.62	0.00	0.01	0.00	0.35	0.00	0.02	0.00
fpppp	0.72	0.00	0.01	0.01	0.22	0.00	0.02	0.03
hydro2d	0.61	0.01	0.10	0.03	0.09	0.00	0.14	0.03
mdljsp2	0.75	0.00	0.02	0.01	0.22	0.00	0.00	0.00
nasa7	0.41	0.02	0.19	0.11	0.22	0.02	0.03	0.01
ora	0.70	0.00	0.00	0.00	0.30	0.00	0.00	0.00
spice	0.64	0.01	0.14	0.10	0.09	0.00	0.01	0.01
su2cor	0.34	0.00	0.25	0.10	0.09	0.00	0.11	0.11
swm256	0.50	0.00	0.16	0.07	0.09	0.00	0.14	0.03
tomcatv	0.47	0.00	0.24	0.07	0.11	0.00	0.08	0.04
wave5	0.56	0.00	0.03	0.02	0.20	0.00	0.11	0.07
compress	0.62	0.00	0.06	0.05	0.25	0.00	0.01	0.00
eqntott	0.86	0.00	0.02	0.02	0.08	0.00	0.01	0.00
espresso	0.76	0.00	0.03	0.01	0.16	0.00	0.02	0.01
gcc	0.63	0.00	0.02	0.02	0.26	0.00	0.04	0.03
li	0.62	0.00	0.02	0.01	0.32	0.00	0.02	0.01
sc	0.57	0.00	0.08	0.05	0.15	0.00	0.08	0.07

XOR-5-5								
Program	Loads				Stores			
	H_f	H_s	M_f	M_s	H_f	H_s	M_f	M_s
APS	0.59	0.04	0.02	0.03	0.27	0.02	0.02	0.01
CSS	0.68	0.05	0.02	0.02	0.17	0.02	0.02	0.01
LGS	0.60	0.04	0.01	0.01	0.25	0.03	0.03	0.03
LWS	0.62	0.06	0.01	0.01	0.23	0.05	0.01	0.01
NAS	0.64	0.03	0.02	0.03	0.22	0.02	0.02	0.02
OCS	0.40	0.02	0.05	0.03	0.34	0.01	0.10	0.05
SDS	0.63	0.06	0.02	0.01	0.23	0.03	0.01	0.01
TFS	0.60	0.04	0.03	0.02	0.16	0.01	0.07	0.06
TIS	0.55	0.06	0.04	0.02	0.29	0.02	0.01	0.00
WSS	0.57	0.08	0.04	0.03	0.19	0.03	0.03	0.03
alvinn	0.69	0.01	0.04	0.01	0.25	0.00	0.00	0.00
doduc	0.70	0.05	0.03	0.02	0.12	0.02	0.03	0.02
ear	0.61	0.01	0.01	0.01	0.34	0.00	0.01	0.01
fpppp	0.60	0.12	0.01	0.01	0.18	0.03	0.02	0.03
hydro2d	0.54	0.09	0.07	0.05	0.08	0.01	0.11	0.06
mdljsp2	0.73	0.02	0.01	0.01	0.22	0.00	0.00	0.00
nasa7	0.36	0.07	0.18	0.12	0.20	0.03	0.02	0.02
ora	0.65	0.04	0.00	0.00	0.28	0.02	0.00	0.00
spice	0.58	0.07	0.13	0.11	0.09	0.00	0.01	0.01
su2cor	0.29	0.05	0.24	0.11	0.08	0.01	0.11	0.12
swm256	0.46	0.05	0.16	0.07	0.09	0.01	0.10	0.07
tomcatv	0.40	0.07	0.21	0.10	0.09	0.02	0.06	0.05
wave5	0.53	0.03	0.03	0.02	0.20	0.01	0.10	0.08
compress	0.62	0.00	0.06	0.05	0.25	0.00	0.00	0.00
eqntott	0.85	0.02	0.02	0.02	0.08	0.01	0.01	0.00
espresso	0.76	0.01	0.03	0.02	0.16	0.00	0.02	0.01
gcc	0.58	0.05	0.02	0.02	0.24	0.02	0.04	0.03
li	0.55	0.07	0.02	0.01	0.28	0.03	0.01	0.01
sc	0.56	0.01	0.08	0.05	0.14	0.01	0.08	0.07

RegNum								
Program	Loads				Stores			
	H_f	H_s	M_f	M_s	H_f	H_s	M_f	M_s
APS	0.56	0.07	0.02	0.03	0.25	0.04	0.02	0.02
CSS	0.64	0.09	0.02	0.02	0.17	0.02	0.02	0.01
LGS	0.56	0.07	0.01	0.01	0.25	0.04	0.03	0.03
LWS	0.61	0.07	0.01	0.01	0.23	0.05	0.01	0.01
NAS	0.63	0.03	0.02	0.03	0.21	0.03	0.02	0.02
OCS	0.39	0.03	0.04	0.04	0.34	0.01	0.08	0.07
SDS	0.63	0.07	0.02	0.01	0.21	0.04	0.01	0.01
TFS	0.60	0.04	0.03	0.02	0.15	0.02	0.06	0.06
TIS	0.42	0.19	0.04	0.02	0.27	0.04	0.01	0.00
WSS	0.53	0.12	0.04	0.03	0.17	0.05	0.03	0.03
alvinn	0.68	0.01	0.03	0.02	0.25	0.00	0.00	0.00
doduc	0.66	0.09	0.03	0.02	0.11	0.02	0.03	0.02
ear	0.60	0.02	0.01	0.01	0.33	0.01	0.01	0.01
fpppp	0.56	0.16	0.01	0.01	0.17	0.04	0.02	0.03
hydro2d	0.56	0.06	0.07	0.05	0.08	0.01	0.10	0.07
mdljsp2	0.71	0.05	0.01	0.01	0.22	0.00	0.00	0.00
nasa7	0.36	0.06	0.16	0.14	0.21	0.03	0.02	0.03
ora	0.64	0.06	0.00	0.00	0.28	0.02	0.00	0.00
spice	0.50	0.15	0.12	0.12	0.09	0.01	0.01	0.01
su2cor	0.26	0.08	0.19	0.15	0.06	0.03	0.11	0.12
swm256	0.40	0.11	0.15	0.08	0.09	0.01	0.09	0.08
tomcatv	0.36	0.11	0.18	0.12	0.09	0.02	0.07	0.05
wave5	0.45	0.11	0.03	0.02	0.18	0.03	0.10	0.08
compress	0.57	0.05	0.06	0.06	0.24	0.01	0.00	0.00
eqntott	0.83	0.03	0.02	0.02	0.07	0.01	0.00	0.00
espresso	0.70	0.06	0.02	0.02	0.15	0.01	0.01	0.02
gcc	0.52	0.11	0.02	0.02	0.22	0.04	0.04	0.03
li	0.48	0.15	0.02	0.02	0.25	0.07	0.01	0.01
sc	0.50	0.07	0.07	0.05	0.13	0.02	0.06	0.09

Proc								
Program	Loads				Stores			
	H_f	H_s	M_f	M_s	H_f	H_s	M_f	M_s
APS	0.57	0.06	0.02	0.03	0.25	0.04	0.02	0.02
CSS	0.67	0.07	0.02	0.02	0.18	0.01	0.02	0.01
LGS	0.59	0.05	0.01	0.01	0.26	0.03	0.03	0.03
LWS	0.62	0.07	0.01	0.01	0.23	0.04	0.01	0.01
NAS	0.64	0.02	0.02	0.03	0.24	0.00	0.02	0.02
OCS	0.39	0.03	0.04	0.04	0.34	0.01	0.08	0.07
SDS	0.63	0.07	0.02	0.01	0.21	0.04	0.01	0.01
TFS	0.59	0.05	0.03	0.02	0.15	0.02	0.06	0.06
TIS	0.39	0.22	0.04	0.02	0.28	0.04	0.01	0.00
WSS	0.51	0.14	0.04	0.03	0.18	0.04	0.03	0.03
alvinn	0.68	0.01	0.03	0.02	0.25	0.00	0.00	0.00
doduc	0.69	0.06	0.03	0.02	0.13	0.01	0.03	0.02
ear	0.60	0.02	0.01	0.01	0.33	0.01	0.01	0.01
fpppp	0.60	0.12	0.01	0.01	0.18	0.03	0.02	0.03
hydro2d	0.53	0.10	0.07	0.05	0.08	0.01	0.10	0.07
mdljsp2	0.71	0.04	0.01	0.01	0.22	0.00	0.00	0.00
nasa7	0.35	0.07	0.16	0.14	0.21	0.02	0.02	0.03
ora	0.67	0.02	0.00	0.00	0.30	0.00	0.00	0.00
spice	0.51	0.14	0.12	0.12	0.09	0.00	0.01	0.01
su2cor	0.28	0.06	0.19	0.16	0.07	0.02	0.11	0.12
swm256	0.40	0.11	0.15	0.08	0.09	0.01	0.09	0.08
tomcatv	0.31	0.16	0.18	0.13	0.09	0.02	0.07	0.05
wave5	0.46	0.10	0.03	0.02	0.18	0.02	0.10	0.08
compress	0.59	0.03	0.06	0.06	0.25	0.01	0.00	0.00
eqntott	0.85	0.02	0.02	0.02	0.08	0.00	0.01	0.00
espresso	0.71	0.05	0.02	0.02	0.15	0.01	0.01	0.02
gcc	0.55	0.08	0.02	0.02	0.23	0.03	0.04	0.03
li	0.52	0.10	0.02	0.02	0.25	0.07	0.01	0.01
sc	0.52	0.05	0.07	0.05	0.14	0.01	0.06	0.09