Inexpensive Implementations of Set-Associativity

R. E. Kessler[†], *Richard Jooss, Alvin Lebeck and Mark D. Hill*[‡]

University of Wisconsin
Computer Sciences Department
Madison, Wisconsin 53706

ABSTRACT

The traditional approach to implementing wide set-associativity is expensive, requiring a wide tag memory (directory) and many comparators. Here we examine alternative implementations of associativity that use hardware similar to that used to implement a direct-mapped cache. One approach scans tags serially from most-recently used to least-recently used. Another uses a partial compare of a few bits from each tag to reduce the number of tags that must be examined serially. The drawback of both approaches is that they increase cache access time by a factor of two or more over the traditional implementation of set-associativity, making them inappropriate for cache designs in which a fast access time is crucial (e.g. level one caches, caches directly servicing processor requests).

These schemes are useful, however, if (1) the low miss ratio of wide set-associative caches is desired, (2) the low cost of a direct-mapped implementation is preferred, and (3) the slower access time of these approaches can be tolerated. We expect these conditions to be true for caches in multiprocessors designed to reduce memory interconnection traffic, caches implemented with large, narrow memory chips, and level two (or higher) caches in a cache hierarchy.

1. Introduction

The selection of associativity has significant impact on cache performance and cost [Smit86] [Smit82] [Hill87] [Przy88a]. The *associativity* (*degree of associativity, set size*) of a cache is the number of places (*block frames*) in the cache where a block may reside. Increasing associativity reduces the probability that a block is not found in the cache (the *miss ratio*) by decreasing the chance that recently referenced blocks map to the same place [Smit78]. However, increased associativity may nonetheless result in longer effective access times since it can increase the latency to retrieve data on a cache *hit* [Hill88, Przy88a]. When it is important to minimize hit times direct-mapped (associativity of one) caches may be preferred over caches with higher associativity.

Wide associativity is important when: (1) miss times are very long or (2) memory and memory interconnect contention delay is significant or sensitive to cache miss ratio. These points are likely to be true for shared memory multiprocessors. Multiprocessor caches typically service misses via a multistage interconnect or bus. When a multi-stage interconnect is used the miss latency can be large whether or not contention exists. Bus miss times with low utilizations may be small, but delays due to contention among processors can become large and are sensitive to cache miss ratio. As the cost of a miss increases, the reduced miss ratio of wider associativity will result in better performance when compared to direct-mapped caches.

Associativity is even more useful for *level two caches* in a two-level multiprocessor cache hierarchy. While the *level one cache* must service references from the processor at the speed of the processor, the level two cache can be slower since it services only processor references that miss in the level one cache. The additional hit time delay incurred by associativity in the level two cache is not as important [Przy88b]. Reducing memory and memory interconnect traffic is a larger concern. Wide associativity also simplifies the maintenance of multi-level inclusion [Baer88]. This is the property that all data contained in lower level caches is contained in their corresponding higher level caches. Multi-level inclusion is useful for reducing coherency invalidations to level one caches. Finally, preliminary models indicate that increasing associativity reduces the average number of empty cache block frames when coherency invalidations are frequent[1]. This implies that wider associativity will result in better utilization of the cache.

Unfortunately, increasing associativity is likely to increase the board area and cost of the cache relative to a direct-mapped cache. Traditional implementations of a-way set-associative caches read and compare all a tags of a set in parallel to determine where (and whether) a given block resides in the cache. With t-bit tags, this requires a tag memory that can provide $a \times t$ bits in parallel. A direct-mapped cache can use fewer, narrower, deeper chips since it requires only a t-bit wide tag memory. Traditional implementations of associativity also use a comparators (each t-bits wide) rather than one, wider data memory, more buffers, and more multiplexors as compared to a direct-mapped cache. This adds to the board area needed for wider associativity. As the size of memory chips increases, it becomes more expensive to consume board area with multiplexors and other logic since the same area could hold more cache memory.

While numerous papers have examined associativity [Lipt68] [Kapl73] [Bell74] [Stre76] [Smit78] [Smit82] [Clar83] [Agar88], most have assumed the traditional implementation. One of the few papers describing a cache with a non-traditional implementation of

[†] This work has been supported by graduate fellowships from the National Science Foundation and the University of Wisconsin-Madison.
[‡] This work was sponsored in part by research initiation grants from the graduate school of the University of Wisconsin-Madison.

[1] A miss to a set-associative cache can fill any empty block frame in the set, whereas, a miss to a direct-mapped cache can fill only a single frame. Increasing associativity increases the chance that an invalidated block frame will be quickly used again by making more empty frames available for reuse on a miss.

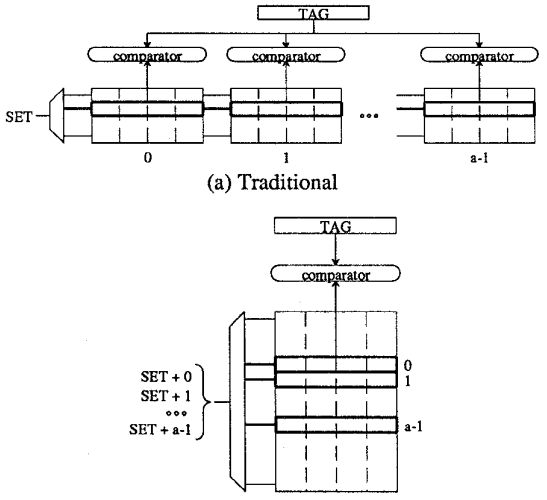

Figure 1. Implementing Set-Associativity.

Part (a) of this figure (top) shows the traditional implementation of the logic to determine hit/miss in an a-way set-associative cache. This logic uses the "SET" field of the reference to select one t-bit tag from each of a banks. Each stored tag is compared to the incoming tag ("TAG"). A hit is declared if a stored tag matches the incoming tag, a miss otherwise.

Part (b) (bottom) shows a serial implementation of the same cache architecture. Here the a stored tags in a set are read from one bank and compared serially (the tags are addressed with "SET" concatenated with 0 through $a-1$).

associativity is [Chan87]. It discusses a cache implemented for a System/370 CPU that has a one-cycle hit time to the most-recently-used (MRU) block in each set and a longer access time for other blocks in the set, similar to the Cray-1 instruction buffers [Cray76] and the biased set-associative translation buffer described in [Alex86].

This paper is about lower cost implementations of associativity, implementations other than the traditional. We introduce cache designs which combine the lower miss ratio of associativity and the lower cost of direct-mapped caches. In the new implementations the width of the comparison circuitry and tag memory is t, the width of one tag, instead of the $a \times t$ required by the traditional implementation. Implementations using tag widths of $b \times t$ $(1 < b < a)$ are possible and can result in intermediate costs and performance, but are not considered here. This paper is not about level two caches per se, but we expect these low cost schemes to be applicable to level two caches. We organize this paper as follows. Section 2 introduces the new approaches to implementing associativity, shows how they cost less than the traditional implementation of associativity, and predicts how they will perform. Section 3 analyzes the approaches of Section 2 with trace-driven simulation.

2. Alternative Approaches to Implementing Set-Associativity

Let a, a power of two, be a cache's associativity and let t be the number of bits in each address tag. During a cache reference, an implementation must determine whether any of the a stored tags in the set of a reference match the incoming tag. Since at most one stored tag can match, the search can be terminated when a match is found (a cache hit). All a stored tags, however, must be examined on a cache miss.

Figure 1a illustrates the traditional implementation of the tag memory and comparators for an a-way set-associative cache, which reads and *probes* all tags in parallel. We define a probe as a comparison of the incoming tag and the tag memory. If any one of the stored tags match, a hit is declared. We concentrate only on cache tag memory and comparators, because they are what we propose to implement differently. Additional memory (not shown) is required by any implementation of associativity with a cache replacement policy other than random. A direct-mapped cache does not require this memory. The memory for the cache data (also not shown) is traditionally accessed in parallel with the tag memory.

Figure 1b shows a naive way to do an inexpensive set-associative lookup. It uses hardware similar to a direct-mapped cache, but serially accesses the stored tags of a set until a match is found (a hit) or the tags of the set are exhausted (a miss). Note how it requires only a single comparator and a t-bit wide tag memory, whereas, the traditional implementation requires t comparators and an $a \times t$ wide tag memory.

Unfortunately, the naive approach is slow in comparison to the traditional implementation. For hits, each stored tag is equally likely to hold the data. Half the non-matching tags are examined before finding the tag that matches, making the average number of probes $(a-1)/2 + 1$. For a miss, all a stored tags must be examined in series, resulting in a probes. The traditional implementation requires only a single probe in both cases.

2.1. The MRU Approach

The average number of probes needed for a hit may be reduced from that needed by the naive approach by ordering the stored tags so that the tags most likely to match are examined first. One proposed order [So88] [Matt70] is from most-recently-used (MRU) to least-recently-used (LRU). This order is effective for level one caches because of the temporal locality of processor reference streams [So88] [Chan87]. We find (in Section 3) that it is also effective for level two caches due to the temporal locality in streams of level one cache misses.

One way to enforce an MRU comparison order is to swap blocks to keep the most-recently-used block in block frame 0, the second most-recently-used block in block frame 1, etc. Since tags (and data) would have to be swapped between consecutive cache accesses in order to maintain the MRU order, this is not a viable implementation option for most set-associative caches.[2] A better way to manage an MRU comparison order, illustrated in Figure 2a, is to store information for each set indicating its ordering. Fortunately, information similar to a MRU list per set is likely to be maintained anyway in a set-associative cache implementing a true LRU replacement policy. In this case there is no extra memory requirement to store the MRU information. We will also analyze (in section 3) reducing the length of the MRU list, using approximate rather than full MRU searches, to further decrease memory requirements. Unfortunately, the lookup of MRU information must precede the probes of the tags[3]. This will lead to longer cache lookup times than would the swapping scheme.

If we assume that the initial MRU list lookup takes about the same time as one probe, the average number of probes required on a cache lookup resulting in a hit using the MRU approach is $1 + \sum_{i=1}^{a} i f_i$ where f_i is the probability the i-th MRU tag matches, given that one of them will match[4]. The MRU scheme performs particularly poorly on cache misses, requiring $1 + a$ probes. This

[2] While maintaining MRU order using swapping may be feasible for a 2-way set-associative cache, Agarwal's hash-rehash cache [Agar87] can be superior to MRU in this 2-way case.

[3] While it is possible to lookup the MRU information in parallel with the level-one-cache access, it is also possible to start level-two-cache accesses early for any of the other implementation approaches [Bren84].

[4] Each f_i is equal to the probability of a reference to LRU distance i divided by the hit ratio, for a given number of sets [Smit78].

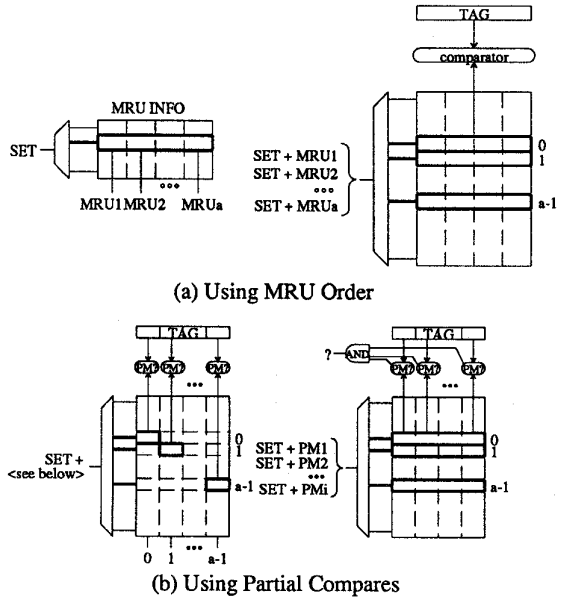

Figure 2. Improved Implementations of Serial Set-Associativity.

Part (a) of this figure (top) shows an implementation of serial set-associativity using ordering information. This approach first reads MRU ordering information (left) and then probes the stored tags from the one most-likely to match to the one least-likely to match (right). Note "+" represents concatenate.

Part (b) (bottom) shows an implementation of serial set-associativity using partial compares. This approach first reads k ($k = \lfloor t/a \rfloor$) bits from each stored tag and compares them with the corresponding bits of the incoming tag. The second step of this approach serially compares all stored tags that partially matched ("PM") with the incoming tag until a match is found or the tags are exhausted (right).

is one more than the naive implementation on misses since the MRU list is uselessly consulted.

2.2. The Partial Compare Approach

We have carefully defined a probe to be the comparison of the incoming tag and the tag memory, without requiring that all bits of the tag memory come from the same stored tag. We now introduce the *partial compare* approach that uses a two step process to often avoid reading all t bits of each stored tag. In step one, the partial compare approach reads t/a bits from each of a stored tags and compares them with the corresponding bits of the incoming tag. Tags that fail this *partial comparison* cannot hit and need not be examined further on a cache lookup. In step two, all stored tags that passed step one are examined serially with t-bit (full) compares.

The implementation of partial compares is not costly, as it can use the same memory and comparators as the naive approach assuming k, the partial compare width ($k = \lfloor t/a \rfloor$), is a multiple of memory chip and comparator width. Partial compares are done with the help of a few tricks. The first trick, illustrated in Figure 2b, is to provide slightly different addresses to each k-bit wide collection of memory chips, addressing the i-th collection with the address of the set concatenated with $\log_2 i$. The second trick is to divide the t-bit comparator into a separate k-bit comparators[5].

[5] If $k \times a$ does not equal t then $\lfloor t/a \rfloor \times a$ bits of the tag can be used for partial compares, with another comparator for the extra bits.

This is straight-forward, since wide comparators are often implemented by logically AND-ing the results of narrow comparators. Note how step two of this partial compare approach uses the same tag memory and comparators as step one, but does full tag compares rather than partial compares.

The performance of this approach depends on how well the partial compares eliminate stored tags from further consideration. For independent tags, the average number of probes will be minimized if each of the values $[0, 2^k - 1]$ is equally likely for each of the k-bit patterns on which partial compares are done. While this condition may be true for physical address tags, it is unlikely to be true for the high order tag bits of virtual addresses. Nevertheless, we can use the randomness of the lower bits of the virtual address tag to make the distribution of the higher ones more uniform and independent. For example, one can transform a tag to a unique other tag by exclusive-oring the low-order k bits of the tag with each of the other k-bit pieces of the tag before it is stored in the tag memory. Incoming tags will go through the same transformation so that the incoming tag and the stored tag will match if the untransformed tags are the same. The original tags can be retrieved from the tag memory for writing back blocks on replacement via the same transformation in which they were stored (i.e. the transformation is its own inverse). This method is used throughout this paper to produce stored tags with better probabilistic characteristics. We will also analyze using no transformation, and using a more sophisticated one in Section 3. We make the assumption in our analysis to follow that each of the values $[0, 2^k - 1]$ is equally likely and independent for each partial compare. Our trace-driven simulation (in Section 3) tests this assumption.

The probability that an incoming tag partially-matches a stored tag is $1/2^k$. A *false match* is a partial tag match which will not lead to a match of the full tag. Given a hit, the expected number of false matches in step one is $(a-1)/2^k$, of which half will be examined in step two before a hit is determined. Thus, the expected number of probes on a hit is $1 + (a-1)/2^{k+1} + 1$, where the terms of the expression are: the probe for the partial comparison (step one), the full tag comparisons (in step two) due to false matches, and the full tag match which produces the hit, respectively. On a miss, the expected number of probes in simply $1 + a/2^k$, the probe for the partial comparison and the number of false matches, respectively.

The partial compare scheme can lead to poor performance if many false matches are encountered in step two. Wider partial compares could eliminate some of these false matches. The partial compare width can be increased by partitioning the a stored tags of a set into s proper *subsets* (each containing a/s tags) and examining the subsets in series[6]. The step one and step two partial compare sequence is performed for each of the subsets to determine if there is a cache hit. The order in which the subsets are examined is arbitrary throughout this paper. Increasing the number of subsets will increase the partial compare width since fewer partial compares are done concurrently. For example, 2 subsets could be used in an 8-way set-associative cache, with 4 entries in each. A lookup in this cache would proceed as two 4-way (single subset) lookups, one after the other. With a 16-bit wide tag memory in this cache, partitioning into 2 subsets would result in 4-bit partial compares. This will result in fewer false matches than with the 2-bit partial compares without subsets. The number of probes per access decreases when using proper subsets if the expected number of false matches is reduced (due to wider partial compares) by more than the number of probes added due to the additional subsets. Subsets may be desirable for implementation considerations in addition to performance considerations if the memory chip or comparator width dictate that the partial compares be wider.

At one extreme (where $s = a$), partial compares with subsets would be implemented as the naive approach, while the other

[6] Note that subsets are not useful with the naive and MRU approaches.

Method	Configuration: Assoc-iativity	Number Subsets	Tag Memory Width (bits)	Expected Probes: Assume Hit	Assume Miss
Traditional	a	1	$a \times t$	1	1
	4	1	64	1	1
Naive	a	1	t	$(1/2)(a-1)+1$	a
	4	1	16	2.5	4
MRU	a	1	t	$1+\sum_{i=1}^{a} i f_i$	$1+a$
	4	1	16	[2, 5]	5
Partial ($k=4$ bits)	a	1	$\max(t, a \times k)$	$2+(a-1)/2^{k+1}$	$1+a/2^k$
	4	1	16	2.09	1.25
Partial w/Subsets	a	s	$\max(t, a/s \times k)$	$2+(1/2)(s-1)+(a-1)/2^{k+1}$	$s+a/2^k$
($k=2$ bits)	8	1	16	2.88	3.00
($k=4$ bits)	8	2	16	2.72	2.5

Table 1. Performance of Set-Associativity Implementations.

For various methods and associativities this table gives the number of subsets, the tag memory width, the number of probes for a hit, and, the number for a miss. The table assumes t-bit tags ($t=16$), k-bit partial compares, and that the i-th most-recently used tag matches with probability f_i on a hit. Note how an increase from 1 to 2 subsets improved the predicted performance of the partial compare approach at an associativity of 8.

($s=1$) can lead to many false matches. An important question to ask is: what number of subsets leads to the best performance (i.e. fewest number of probes per cache lookup) ? The next three answers to this question vary from the most-accurate to the most succinct. (1) One can compute the expected number of probes for each of $s = 1, 2, 4, \ldots, a/2$ and a using the equations for a hit and miss (from Table 1) weighted to reflect your expected miss ratio and choose the minimum. (2) One can ignore misses (which are less common and never require more than twice the probes of hits), assume variables are continuous, and find the optimum partial compare width, $k_{opt} = \log_2 t - 1/2$ for hits only. The optimum number of subsets for hits and misses together is likely to be the value for s resulting from a partial compare width of $\lfloor k_{opt} \rfloor$ or $\lceil k_{opt} \rceil$. (3) Finally, one can observe that many tags in current caches are between 16 and 32 bits wide, implying the number of subsets that gives at least four-bit partial compares will work well.

Table 1 summarizes our analysis of the expected number of probes required for the traditional, naive, MRU and partial compare approaches to implementing set-associativity. Note that this table as well as most of the trace-driven simulation assumes 16 bit tags are used. We will examine the positive effect of increasing the tag width on the partial compare approach in section 3.

Table 2 summarizes paper implementations of tag memory and comparison logic for a direct-mapped cache, a traditional implementation of set-associativity, and an implementation of set-associativity using MRU and partial compares. We found that the MRU and partial compare implementations have a slower access time than the traditional implementation of associativity but includes no implementation surprises. Most notably, the control logic was found to be of reasonable complexity. The MRU and partial compare implementations use hardware similar to a direct-mapped cache and can make effective use of page-mode dynamic RAMs, as would other serial implementations of set-associativity. Page-mode dynamic RAMs are those in which the access time of multiple probes to the same set can be significantly less than if the probes were to other sets. Subsequent probes take less than half the time of the first probe to the set. Cache cost is reduced in two ways when using one of the alternative implementations of associativity. First, tag memory cost is directly reduced, by 1/3 to 1/2 in our design. Second, cache data memory cost is reduced since only 1, rather than a words, need to be read at a time.

3. Trace-Driven Performance Comparison

This section analyzes the performance of the low-cost schemes to implement set-associativity in level two caches using simulation with relatively short multiprogramming traces. We analyze associativity in the level two cache since the low cost implementations of associativity are more appropriate for level two (or higher) caches than for level one caches. We concentrate on presenting and characterizing the relative performance of the alternatives. We do not demonstrate the absolute utility of these approaches to important future cache configurations (e.g. multiple megabyte level two caches in multiprocessors) since our traces are for a single processor and are not sufficiently long to exercise very large caches.

The makeup of the traces and the assumed cache configurations are indicated in Table 3. We assume a uniprocessor system with a two level cache hierarchy (a level one cache and a level two cache) in our study, largely because the traces were

Cache Tag Memory and Comparator Implementations								
	Using Dynamic RAMs				Using Static RAMs			
	Direct-Mapped	4-Way Set-Associative			Direct-Mapped	4-Way Set-Associative		
		Traditional	MRU	Partial		Traditional	MRU	Partial
Memory Packages								
Basic Access Time (ns)	100	80	100	100	40	40	40	40
Page Mode Access Time (ns)	n/a	n/a	35	35	n/a	n/a	n/a	n/a
Basic Cycle Time (ns)	190	160	190	190	40	40	40	40
Page Mode Cycle Time (ns)	n/a	n/a	35	35	n/a	n/a	n/a	n/a
Size (bits)	1Mx8	256Kx8	1Mx8	1Mx8	1Mx4	256Kx(16,8)	1Mx4	1Mx4
Implementations								
Access Time (ns)	136	132	150+50x	150+50y	61	84	65+55x	65+55y
Cycle Time (ns)	230	190	250+50(x+u)	250+50y	85	100	75+55(x+u)	75+55y
Number of Packages	18	42	22	21	20	37	25	24

Table 2. Trial Set-Associativity Implementations.

This table compares paper implementations of the tag memory and comparison logic for a direct-mapped and four-way set-associative cache holding 1 million 24-bit tags, assuming dynamic or static RAM chips housed in hybrid packages. The top half of the table summarizes the memory packages used to implement tag memory, while the bottom half gives cache implementation numbers. The MRU implementation assumes that the MRU list storage costs nothing extra (as it would if full LRU replacement is used). MRU access and cycles are given assuming "x" is the expected number of probes after reading the MRU information ("x" is between 1 and a for hits, a for misses) and "u" is the probability that MRU information must be updated. Partial compare access and cycles are given assuming "y" probes in step two ("y" is between 1 and a for hits and 0 and a for misses). The number of packages assumes some semi-custom logic and hybrid packages.

Trace-Driven Two-Level Cache Simulation	
Traces	ATUM [Agar86] virtual address traces of a multiprogrammed operating system, described in [Hill87]. Operating system references are included as well as references of user-level processes. One very large trace (over 8 million references) was constructed as a concatenation of 23 individual ATUM traces, each of which is approximately 350,000 references. Cache flushes of the level one and level two caches were inserted between each of the 23 traces, thus each trace starts from a "cold" cache.
Level One Cache	A direct-mapped write-back cache. On misses causing replacement of a dirty block, the new block is first obtained via a *read-in* request, then a *write-back* is issued to the level two cache. Three level one caches are simulated: A 4 Kbyte cache with a 16 byte block size (4K-16); 16 Kbyte with 16 byte blocks (16K-16); and 16 Kbyte with 32 byte blocks (16K-32). The miss ratios corresponding to these level one caches are: 0.1181, 0.0657, and 0.0513, respectively.
Level Two Cache	An a-way set-associative write-back cache which services read-ins and write-backs from the level one cache. We compare different implementations of associativity in the level two cache. The least-recently-used entry in a set is replaced on a cache miss. We simulate five different level two caches: a 64 Kbyte cache with 16 byte block size (64K-16); 64 Kbyte with 32 byte blocks (64K-32); 256 Kbyte with 16 byte blocks (256K-16); 256 Kbyte with 32 byte blocks (256K-32); and 256 Kbyte with 64 byte blocks (256K-64). While multi-level inclusion is not enforced in this simulation, by monitoring the number of write-backs which missed when written back to the level two cache we were able to extrapolate that the maintenance of multi-level inclusion would have a very small effect (in most configurations studied, no effect) on the miss ratio of the level two cache (and no effect on the miss ratio of the level one cache).

Table 3. Detailed Information on the Trace-Driven Simulation.

uniprocessor traces. The level one cache is direct-mapped, while the level two cache is of varying associativity. Both caches are write-back caches, with the level two cache servicing *read-in* and *write-back* requests from the level one cache. We chose this write-back configuration to minimize the amount of communication between cache levels. This can be important in a shared memory multiprocessor since the level one cache will be utilized servicing processor references while the level two cache is servicing coherency invalidations, as in [Good88]. Also, it was found in [Shor88] that this configuration has better performance than if either cache is write-through. Cache sizes simulated here (up to 256 Kbytes) are limited by the size of the traces. We expect future level two (and higher) caches to be considerably larger (e.g. 4 Mbytes). Though the results presented are for "cold" caches, limited "warmer" results were found to be similar, except that the miss ratios were smaller.

The graphs in Figure 3 show the average number of probes versus the associativity of the level two cache for a 16K-16 (16 Kbyte capacity with 16 byte block size) level one cache and 256K-32 (256 Kbyte with 32 byte block size) level two cache. The tag width is 16-bits ($t = 16$) and the partial compare width 4-bits ($k = 4$) in all simulations unless otherwise specified. 1, 2, and 4 subsets were used for 4, 8, and 16-way set-associative partial compare implementations, respectively. The graphs indicate the general linearly increasing relationship between the number of probes required per search and the associativity. The number of probes per access is expected to increase for the alternative implementations of associativity as the associativity increases since there are more places where a given cache block can reside. A cache lookup simply must look in more places on the average. For wider associativity to be preferred, the added delay for these additional probes must be more than offset by the time saved servicing fewer misses. One would also expect the Naive and Partial schemes to have a linear relationship between probes per access and associativity. However, the fact that this relation is linear for MRU came as a surprise. We will examine the MRU and partial schemes more closely in subsequent figures. As will always be the case, the traditional implementation of associativity results in the minimum number of probes. These graphs show that the partial compare approach performs the best of the low cost implementations. The naive scheme performs the worst, with the MRU scheme between them.

Figure 3 also shows the performance benefit of a write-back optimization which can be made when the multi-level inclusion property is maintained with a cache hierarchy. The level one cache can be certain that all write-back requests will hit in the level two cache. It can also be certain that the block will reside in precisely the same position in which it was loaded in the level two cache from memory (if blocks do not change position in the level two cache from the time they are loaded to the time they are replaced). This implies that if the level one cache retains a $\log_2 a$-bit indicator of which position in the set the given cache block occupies (a is the associativity of the level two cache), write-backs can proceed without requiring tag probes. Note that even if multi-level inclusion is not maintained, the indicators in the level one cache can be used as *hints*, not always correct, where the entry resides in the level two cache.

All the methods, Traditional, Naive, MRU, and Partial require no probes to service a write-back request when using the write-back optimization. Since write-backs are approximately 20% of the requests to the level two cache (as shown in Table 4), this can result in significant performance improvements, as indicated in the figure. We feel the cost of implementing this optimization is sufficiently modest (2 bits per level one cache entry for a 4-way set-associative level two cache) to warrant its use when implementing one of the reduced cost implementations of associativity. We assume the write-back optimization is used, and all subsequent figures contain data for read-in requests only, since the different approaches perform the same on write-backs. Write-back requests are still considered references as they update the MRU list, determining the replacement policy of the cache.

Table 4, presented at the end of the paper, lists the number of probes required for various cache configurations when using the naive, MRU, and partial schemes. Note that the data in Table 4 assumes the write-back optimization is being used. Table 4 uses the terms *global miss ratio* and *local miss ratio* [Przy88b]. The global miss ratio is the fraction of processor requests which miss in both the level one and level two cache. The local miss ratio of the level two cache is the fraction of read-ins and write-backs from the level one cache which miss in the level two cache. Note that 8 and 16-way set-associativity did not improve the miss ratios substantially over 4-way in our simulations.

The partial scheme performs best (requires the least number of probes) for most configurations studied. However, MRU did perform best for the configuration with the largest level two cache block size and the largest ratio of level two to level one cache size (4K-16 256K-64). This leads to several key observations regarding the MRU scheme. First, MRU is a better scheme as the block size of the level two cache increases relative to the level one cache block size. MRU can take advantage of the larger block sizes in the level two cache since more data spatially near the latest reference is in the MRU block. Second, its performance improves as the size of the level one cache is decreased relative to the size of the level two cache. The miss stream from a smaller level one cache has more temporal locality than larger level one caches. This locality results more often in hits to the first entry in the MRU list of a set.

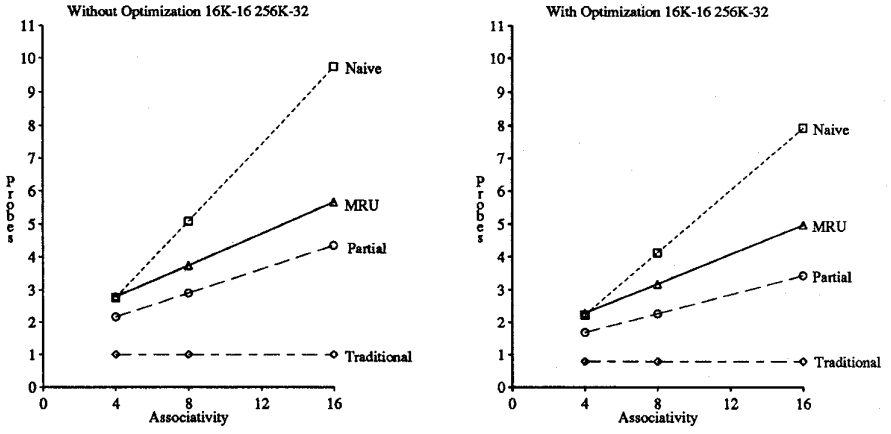

Figure 3. Probes for Read-Ins and Write-Backs.

This figure shows the average number of probes per cache access for the Traditional, Partial, MRU, and Naive implementations of various associativities. It also shows the usefulness of the write-back optimization in which the first level cache retains an indicator which allows it to write-back to the second level cache without any tag comparisons (probes).

The number of probes per cache access increases with associativity for the non-traditional implementations since there are more places for a given block to reside. Lower effective access times may nevertheless result, particularly as miss latencies are increased, since higher associativity results in lower miss ratios.

Figure 4 compares the performance of the schemes on read-in hits and misses separately. It shows how the partial and MRU approach are close in performance on hits, followed by the naive approach. The partial approach is the undeniable winner on misses, dominating the a and $a+1$ probes needed by the naive and MRU approaches, respectively[7]. The rest of the figures in this paper will concentrate on read-in hits for that reason. One should keep in mind, however, that the figures will be biased in favor of the MRU and naive approaches, when compared to the partial approach.

Figure 5 looks more closely at the MRU scheme. It examines the performance impact of shortening the MRU list to less than the total number of entries in a set. An associative lookup with a shortened MRU list proceeds by first searching the entries in the list in order and then searching the rest of the set in an arbitrary order. The examination shows that it is not necessary to retain the entire MRU list to achieve close to the performance of the entire list. It also shows that the length of the shortened MRU list must increase linearly with associativity to achieve near the performance of a full MRU list. For instance, a reduced MRU list of two entries performs well for an associativity of 8, whereas, a reduced list of 4 entries is needed to produce comparable performance with an associativity of 16.

The right graph in figure 5 plots the values of f_i for various associativities in the level two cache. Lower associativities result in a higher probability that a hit is to the first entry of the MRU list. For instance, the probability is 75%, 60%, and 36% for 4, 8, and 16-way associativities, respectively, in the right graph of Figure 5. It was found in [So88] that the probability that a hit is to the first element in the MRU list of a 4-way set-associative level one cache is above 90% for cache sizes greater than 32 Kbytes (block size = 128 bytes). We have not seen this percentage reach 90% for the level two cache in any of our cache configurations. The closest was 89% with the 4K-16 level one cache and the 256K-64 4-way set-associative level two cache.

It was previously noted that the linear relationship between the average number of probes per cache access and the associativity of the level two cache was unexpected when using the MRU scheme. This relationship can be explained, with some approximations, by examining the right graph of Figure 5. If the lines in the right graph were straight lines, there would be an exponential (more precisely, geometric) relationship between the probability of a hit and the MRU distance. If this were the case and the slope of these lines (ignoring the log scale of the vertical axis and considering it a linear scale) is proportional to $-1/a$, then, (with some approximations) we can say that there will be a linear relationship between probes and associativity. Since both the conditions are roughly true, it can explain the linearity.

Figure 6 examines the partial compare approach in more detail. It shows that wider tags improve the performance of the partial scheme on read-in hits. The larger tag size allows for a reduced number of subsets in the 8 and 16-way set-associative caches and an increase in the partial compare width for the 4-way set-associative cache. Tag widths may be larger because the system supports a large virtual address space or may be artificially increased for better performance. Note that the number of probes required by the naive and MRU schemes do not change as the tag width is changed.

Figure 6 compares the performance of the partial scheme to the predictions of the theory of Section 2. It shows that the simple transformation outlined in Section 2 in which the low order k bits are exclusive-ored with each of the higher order bits performs worse than the prediction of theory (particularly with 32 bit tags). This is not surprising since the theory is a probabilistic lower bound. We considered other transformations which exclusive-or a bit with a subset of the other bits of the tag. This transformation may be required to be efficiently invertible. If we restrict the bits that are exclusive-ored to be from less significant fields, the resulting transformation produces unique and invertible tags[8]. The improved transformation passes the least significant k-bit field unchanged, exclusive-ors the second least significant field with the

[7] Note that the local miss ratio of large level two caches is not vanishingly small, especially with a large level one cache [Przy88b].

[8] Taking "exclusive-or" as addition and "and" as multiplication, the set {0,1} forms a finite field, denoted by GF(2). Our hash function is a linear transformation T from $GF(2)^t$ to itself, given by a lower-triangular matrix with 1's on the diagonal. It can be shown using Gaussian elimination that T is invertible, and its inverse is lower-triangular as well. See [Pete72] for an introduction to finite fields and linear transformations.

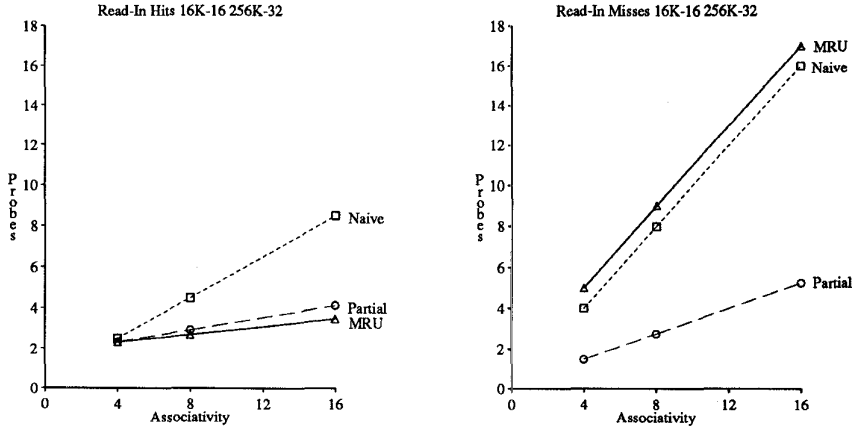

Figure 4. Probes for Read-In Hits and Misses.

This figure separates the performance of the Naive, Partial, and MRU algorithms for read-in hits (on the left) and misses. For hits, the Partial and MRU algorithms perform well, with Naive considerably worse. On misses, the Partial algorithm is superior, followed by the Naive and MRU algorithms. Both the Naive and MRU algorithms cycle through the entire set on a miss, with MRU charged an extra probe for the look-up of the ordered list.

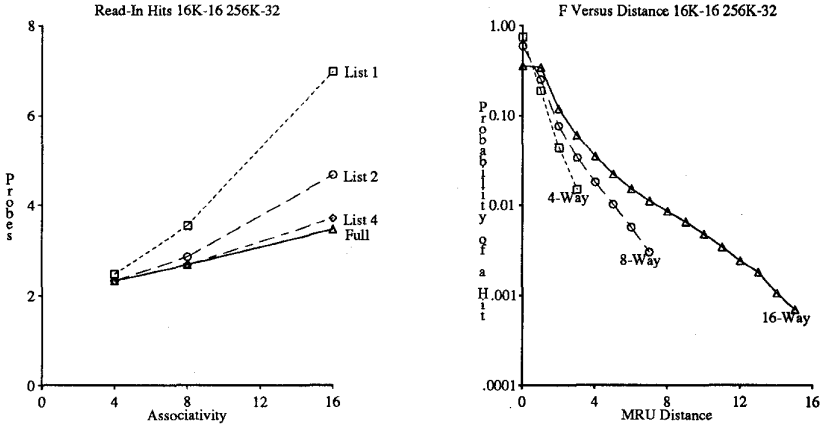

Figure 5. Reduced MRU Lists and Distance Distribution.

This figure demonstrates the performance of the MRU scheme on read-in hits in more detail. The left graph compares the performance of reduced MRU lists. The right graph shows the MRU distance distributions for hits.

first, and exclusive-ors all other fields with both the first and the second fields. The new transformation can be implemented with one two-input exclusive-or gate per higher order bit, the same number required for the original transformation. Unfortunately, the new transformation is not its own inverse, but, the inverse also requires the same number of exclusive-or gates. The left graph of Figure 6 shows that the new transformation results in better performance, particularly for 32-bit tags. This indicates that the transformation should be carefully chosen. We also investigated a transformation in which the bits of the tag are swapped so that the low order bits of the incoming tag are always compared with the low order bits of the stored tag. Its performance was good, near the theory lines in Figure 6, but it is more expensive to implement.

4. Conclusions

We have described and analyzed three methods for implementing set-associative caches which retain many of the implementation advantages of direct-mapped caches while providing the reduced miss ratio of associative cache lookups. These implementations are less expensive than the traditional approach since they eliminate comparators and obviate the need to access cache tags and data within the same set in parallel. Our trace-driven analysis of these schemes for use in level two caches was done using various level one and level two cache configurations. This allowed us to examine the trends of the various schemes as the cache parameters were varied. This is important since the traces were inadequate to simulate the multi-megabyte level two caches we expect will be useful in future systems.

The three low cost schemes explained in this paper are the naive, MRU, and partial compare implementations of set-associativity. The naive scheme uses a linear scan over all the stored tags in a set during a cache lookup. The MRU list scheme

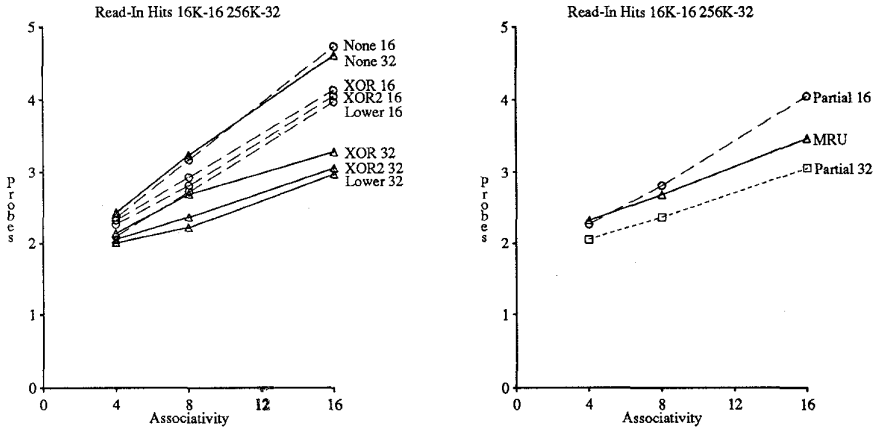

Figure 6. Partial Algorithm With Larger Tags and Different Transformation.

This figure analyzes the performance of the partial scheme on read-in hits in more detail. The left graph compares its performance versus the prediction of theory outlined in Section 2 for 16-bit (dashed lines) and 32-bit tags (solid lines). There are four lines for each tag width: the top line is the results when using no transform (None), the next lower line is the simple transformation of Section 2 (XOR), the next lower line the more sophisticated transformation outlined in Section 3 (XOR2), and the bottom line is the prediction of the theory, a probabilistic lower bound (Lower).

The right graph compares the performance of the partial scheme using the more sophisticated transformation versus the MRU scheme for 16 and 32 bit tags.

retains an ordered list per set to search the stored tags in an "intelligent" order. The partial compare scheme looks once at small pieces of many of the stored tags of a set. It then decides whether it should do full tag comparisons on the tags depending on the outcome of the partial comparison. Naive and partial lookups require the same memory and comparison logic as a direct-mapped cache, only extra control logic is needed for associativity. The MRU scheme may require the extra memory to hold the ordered search list as well as extra hardware to maintain it, although the same hardware is likely needed to implement an LRU cache replacement policy.

The average number of probes (tag memory reads and compares) per cache lookup was measured. As expected, the naive scheme performed poorly as compared to the MRU and partial compare schemes for associativities of 4 and above. Both the MRU and partial compare schemes perform well on cache hits, with perhaps a slight advantage to MRU. The partial compare scheme achieves superior performance on cache misses since it does not require a probe to examine each and every tag in the set.

Over the widest range of cache configurations considered, the partial compare algorithm required the least number of probes per cache access. However, the partial compare scheme is not the best scheme for all cases. The MRU scheme is better when the local miss ratio of the level two cache is small. This is true when the ratio of level two to level one block sizes is large (4 or more) and when the ratio of level one to level two cache sizes is large (64 or more). The partial compare scheme is better when the tag width is increased and when the local miss ratio of the level two cache is increased. The local miss ratio of the level two cache increases when the above cache and blocksize ratios decrease.

This study does not show either the MRU or partial compare schemes to be the superior low cost implementation of associativity for future level two caches. The optimum scheme depends on all the factors above, in particular: the cache size ratio, block size ratio, and the tag width. Since we expect tag widths to be larger than the 16 bits used in most of our study, we favor the partial compare scheme. However, it may be true that the level two to level one cache size ratio will be larger in the future than our simulation, in which case the MRU scheme is more favorable.

We feel that low cost implementations of associativity are useful, particularly for level two caches. The slower access times of the associativity implementations outlined in this paper are less important in level two caches since the processor sees the latency of the level two cache only on a level one cache miss. The lower cost and board area minimization of the approaches presented in this paper may prove to be more important than speed since we expect future level two caches to be large (megabytes). Some recently proposed multiprocessors [Wils87] [Good88] promise to require many large caches. In this environment, cost can be an extremely important consideration.

5. Acknowledgements

The authors would like to thank Jim Smith for some suggestions inspiring this research, Eric Bach for much needed mathematical assistance, and all those involved with the Multicube project for providing focus and feedback. We would also like to thank Anant Agarwal, Mark Horowitz, Richard Sites, and Digital Equipment Corporation for providing the traces used in our study. Thanks also to those who read and improved drafts of this paper: Tom Bricker, Garth Gibson, Ross Johnson, B. Narendran, Harold Stone, Chuck Thacker, Phil Woest, David Wood, and The University of Washington Architecture Lunch Group.

6. References

[Agar86] A. Agarwal, R. L. Sites and M. Horowitz, ATUM: A New Technique for Capturing Address Traces Using Microcode, *Proc. Thirteenth International Symposium on Computer Architecture* (June 1986).

[Agar87] A. Agarwal, Analysis of Cache Performance for Operating Systems and Multiprogramming, Ph.D. Thesis, Technical Report No. CSL-Tech. Rep.-87-332, Stanford University (May 1987).

[Agar88] A. Agarwal, M. Horowitz and J. Hennessy, Cache Performance of Operating Systems and Multiprogramming Workloads, *ACM Trans. on Computer Systems*, 6, 4 (November 1988).

[Alex86] C. Alexander, W. Keshlear, F. Cooper and F. Briggs, Cache Memory Performance in a UNIX Environment, *Computer Architecture News*, 14, 3 (June 1986), 14-70.

[Baer88] J. Baer and W. Wang, On the Inclusion Properties for Multi-Level Cache Hierarchies, *15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii (June 1988).

4-Way Set-Associative Level Two Cache										
Configuration	Global Miss Ratio	Local Miss Ratio	Fraction Write-Back	Naive Probes		MRU Probes		Partial Probes		
				Hits	Total	Hits	Total	Hits	Misses	Total
16K-16 256K-32	0.0143	0.1721	0.2141	1.85	2.22	**1.72**	2.29	**1.72**	**1.49**	***1.68**
16K-16 256K-16	0.0223	0.2665	0.2141	1.77	2.37	1.71	2.59	**1.68**	**1.51**	***1.64**
16K-32 256K-32	0.0144	0.2157	0.2302	1.77	2.25	1.72	2.42	**1.64**	**1.49**	***1.61**
4K-16 256K-64	0.0097	0.0653	0.2083	1.94	2.08	**1.66**	1.88	1.78	**1.46**	***1.76**
4K-16 256K-32	0.0144	0.0964	0.2083	1.92	2.12	**1.66**	1.98	1.81	**1.49**	***1.78**
4K-16 256K-16	0.0223	0.1494	0.2083	1.89	2.20	**1.66**	2.16	1.82	**1.51**	***1.77**
4K-16 64K-32	0.0195	0.1310	0.2083	1.90	2.18	1.84	2.25	**1.62**	**1.29**	***1.58**
4K-16 64K-16	0.0279	0.1870	0.2083	1.86	2.26	1.90	2.48	**1.60**	**1.29**	***1.54**

8-Way Set-Associative Level Two Cache										
Configuration	Global Miss Ratio	Local Miss Ratio	Fraction Write-Back	Naive Probes		MRU Probes		Partial Probes		
				Hits	Total	Hits	Total	Hits	Misses	Total
16K-16 256K-32	0.0141	0.1682	0.2141	3.34	4.12	**1.99**	3.17	2.17	**2.74**	***2.27**
16K-16 256K-16	0.0220	0.2629	0.2141	3.19	4.45	**2.04**	3.87	2.12	**2.76**	***2.28**
16K-32 256K-32	0.0141	0.2110	0.2302	3.18	4.20	**2.07**	3.53	2.08	**2.74**	***2.22**
4K-16 256K-64	0.0094	0.0631	0.2083	3.50	3.78	**1.80**	***2.25**	2.25	**2.72**	2.28
4K-16 256K-32	0.0141	0.0943	0.2083	3.47	3.89	**1.80**	2.48	2.27	**2.74**	***2.32**
4K-16 256K-16	0.0220	0.1473	0.2083	3.40	4.08	**1.82**	2.88	2.27	**2.76**	***2.34**
4K-16 64K-32	0.0189	0.1264	0.2083	3.43	4.00	2.25	3.10	**2.08**	**2.51**	***2.13**
4K-16 64K-16	0.0270	0.1808	0.2083	3.36	4.20	2.42	3.61	**2.04**	**2.51**	***2.12**

16-Way Set-Associative Level Two Cache										
Configuration	Global Miss Ratio	Local Miss Ratio	Fraction Write-Back	Naive Probes		MRU Probes		Partial Probes		
				Hits	Total	Hits	Total	Hits	Misses	Total
16K-16 256K-32	0.0139	0.1663	0.2141	6.31	7.93	**2.58**	4.97	3.07	**5.27**	***3.43**
16K-16 256K-16	0.0218	0.2612	0.2141	6.03	8.64	**2.75**	6.47	2.96	**5.26**	***3.56**
16K-32 256K-32	0.0139	0.2086	0.2302	6.03	8.11	**2.81**	5.77	2.95	**5.27**	***3.43**
4K-16 256K-64	0.0092	0.0619	0.2083	6.61	7.19	**2.08**	***3.00**	3.22	**5.27**	3.35
4K-16 256K-32	0.0139	0.0932	0.2083	6.55	7.43	**2.11**	3.50	3.22	**5.27**	***3.41**
4K-16 256K-16	0.0218	0.1464	0.2083	6.43	7.83	**2.18**	4.35	3.19	**5.26**	***3.49**
4K-16 64K-32	0.0185	0.1240	0.2083	6.48	7.66	**3.04**	4.77	**3.04**	**5.02**	***3.28**
4K-16 64K-16	0.0266	0.1780	0.2083	6.35	8.07	3.45	5.87	**2.99**	**5.03**	***3.35**

Table 4. Data for Various Cache Configurations.

These tables show the number of probes for the different schemes in varying cache configurations, as well as the miss ratios for the configurations. The numbers of probes are shown for hits and totals (hits and misses together). Misses are not shown for the MRU and Naive schemes since they require *associativity*+1 and *associativity* probes, respectively. The bold entries indicate the best method for hits, misses, and in total for the given configurations. The entries with asterisks indicate the best method in total. Write-backs from the level one cache are assumed to require no probes due to the write-back optimization, yet they are counted as a hit and included in the averages. The fraction of write-backs is the fraction of requests from the level one cache which are write-backs.

[Bell74] J. Bell, D. Casasent and C. G. Bell, An Investigation of Alternative Cache Organizations, *IEEE Trans. on Computers*, C-23, 4 (April 1974), 346-351.

[Bren84] J. G. Brenza, Second Level Cache Fast Access, *IBM Technical Disclosure Bulletin*, 26, 10B (March 1984), 5488-5490.

[Chan87] J. H. Chang, H. Chao and K. So, Cache Design of a Sub-Micron CMOS System/370, *14th Annual International Symposium on Computer Architecture*, Pittsburgh, PA (June 1987), 208 - 213.

[Clar83] D. W. Clark, Cache Performance in the VAX-11/780, *ACM Trans. on Computer Systems*, 1, 1 (February, 1983), 24 - 37.

[Cray76] Cray Research Inc., The Cray-1 S Series Hardware Reference Manual, Publication No. HR-0808 (1976).

[Good88] J. R. Goodman and P. J. Woest, The Wisconsin Multicube: A New large-Scale Cache-Coherent Multiprocessor, *Proc. Fifteenth Symposium on Computer Architecture* (June 1988).

[Hill87] M. D. Hill, Aspects of Cache Memory and Instruction Buffer Performance, Ph.D. Thesis, Computer Science Division Technical Report UCB/Computer Science Dept. 87/381, University of California, Berkeley (November 1987).

[Hill88] M. D. Hill, A Case for Direct-Mapped Caches, *IEEE Computer*, 21, 12 (December 1988), 25-40.

[Kapl73] K. R. Kaplan and R. O. Winder, Cache-based Computer Systems, *Computer*, 6, 3 (March, 1973).

[Lipt68] J. S. Liptay, Structural Aspects of the System/360 Model 85, Part II: The Cache, *IBM Systems Journal*, 7, 1 (1968), 15-21.

[Matt70] R. L. Mattson, J. Gecsei, D. R. Shutz and I. L. Traiger, Evaluation techniques for storage hierarchies, *IBM Systems Journal*, 9, 2 (1970), 78 - 117.

[Pete72] W. W. Peterson and E. J. Weldon, Jr., *Error-Correcting Codes*, MIT Press, (1972).

[Przy88a] S. Przybylski, M. Horowitz and J. Hennessy, Performance Tradeoffs in Cache Design, *15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii (June 1988).

[Przy88b] S. A. Przybylski, Performance-Directed Memory Hierarchy Design, Ph.D. Thesis, Technical Report No. CSL-Tech. Rep.-88-366, Stanford University (September 1988).

[Shor88] R. T. Short and H. M. Levy, A Simulation Study of Two-Level Caches, *15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii (June 1988).

[Smit78] A. J. Smith, A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory, *IEEE Trans. on Software Engineering*, SE-4, 2 (March 1978), 121-130.

[Smit82] A. J. Smith, Cache Memories, *Computing Surveys*, 14, 3 (September, 1982), 473 - 530.

[Smit86] A. J. Smith, Bibliography and Readings on CPU Cache Memories and Related Topics, *Computer Architecture News* (January 1986), 22-42.

[So88] K. So and R. N. Rechtschaffen, Cache Operations by MRU Change, *IEEE Trans. on Computers*, C-37, 6 (June 1988).

[Stre76] W. D. Strecker, Cache Memories for PDP-11 Family Computers, *Proc. Third International Symposium on Computer Architecture* (January 1976), 155-158.

[Wils87] A. W. Wilson, Hierarchical Cache/Bus Architecture for Shared Memory Multiprocessors, *14th Annual International Symposium on Computer Architecture*, Pittsburgh, PA (June 1987).