A CASE FOR DIRECT-MAPPED CACHES

by

Mark D. Hill

Computer Sciences Technical Report #778

June 1988

A Case for Direct-Mapped Caches

Mark D. Hill

Computer Sciences Department
University of Wisconsin
Madison, Wisconsin 53706
markhill@cs.wisc.edu

A datum can reside in at most one place in a direct-mapped cache, but in one of n places in an n-way set-associative cache (where n is usually 2, 4, 8 or 16). A direct-mapped organization facilitates a fast, inexpensive lookup, but offers no flexibility regarding what datum may be replaced when it is necessary to make room for a new one. A set-associative organization provides flexibility for replacement, but requires a slower, more expensive lookup that must examine n places in parallel. Thus, direct-mapped caches reduce the time spent handling cache hits (the datum is in the cache), while set-associative caches reduce the time spent on cache misses (the datum is not present).

A set-associative organization is preferred in most current caches, which are relatively-small ($\leq$ 16K bytes) and miss often. For a variety of reasons, however, caches are getting larger. Here I show that a direct-mapped organization is preferred in most large caches ($\geq$ 64K bytes), which rarely miss.

1. Introduction

A *cache* is small, fast buffer in which a system tries to keep those parts of the contents of a larger, slower memory that will be used soon. The purpose of a cache is to improve system cost-performance by providing the capacity of the large, slow memory with an average access time close to that of the small, fast cache. This is possible only if most memory references can be serviced rapidly by the cache without the intervention of the slower memory. Usually the cache is successful, due to *temporal* and *spatial locality,* two properties of most real reference streams [Denn70]. Temporal locality means future references are likely to be made to the same locations as recent requests, while spatial locality suggests that future references are also likely to be made to locations near recent references. Caches take advantage of temporal locality by retaining recently-referenced information, while they exploit spatial locality by loading and retaining (blocks of) information surrounding recent references.

A *CPU cache* is a cache of main memory [Smit82]. Like caches in general, CPU caches are faster and smaller than the memory they buffer. CPU caches are usually five to 20 times faster and 50 to 1000 times smaller than main memory. Because CPU caches must be extremely fast, they are managed entirely by hardware, and for this reason, CPU cache access and management policies must be relatively simple. Table 1 provides some succinct definitions for the basic CPU cache terminology used throughout this paper. Other terms will be defined the first time they are used.

Term	Definition
reference	A request by the processor to read or write a memory location. (synonyms: request, access, processor reference, memory reference)
cache	A small, fast memory that holds active parts of a larger, slower memory. The capacity of a cache is the *cache size*. (synonym: buffer)
hit, miss	References found in the cache are said to *hit;* those not found to *miss*.
memory	A larger, slower memory that provides data on cache misses.
block frame	A location in the cache that holds cached data, an associated address tag and state bits. The address tag gives the main memory address of data held in the block frame. The state bits indicate whether data in valid (not random bits) and can indicate whether data is dirty (must be written to main memory on replacement) or its status in a multiprocessor cache coherency protocol. The capacity of a block frame is the *block size*. (synonym: block)
block	Data from memory that fills a block frame. (synonyms: line, sector)
placement algorithm	The method used to determine where a block may reside in a cache; often selects the *set* of a reference. (synonyms: placement policy, cache organization)
set	A collection of block frames in which a block can reside. (synonym: congruence class)
associativity	The number of block frames in each set. (synonyms: set size, degree of associativity)
n-way set-associative	A placement algorithm that divides a cache's block frames into more than one set of n block frames each (associativity n), where n is greater than one.
direct-mapped	A placement algorithm with single-block sets (associativity one). (synonym: one-way set-associative)
fully-associative	A placement algorithm with one set (associativity c where c is the number of block frames in a cache).
replacement algorithm	The method used to determine which block to replace when a new block is loaded. With set-associative placement, only blocks that reside in the set of the new block are considered for replacement. (synonym: replacement policy)
LRU	A commonly-used replacement algorithm that replaces the *least recently used* block, that is, the one least recently referenced.

Table 1. Selected Basic CPU Cache Terminology.

CPU caches have been studied extensively (for a bibliography see [Smit86]), because properly designed CPU caches have proven effective at increasing system performance, lowering system cost or both. CPU caches continue to be worth studying, because their importance to system cost-performance is increasing and because technological improvements are altering the characteristics of well-designed CPU caches. Since this paper examines only CPU caches, I will often use *cache* instead of *CPU cache*.

The important cache design parameter this paper examines is *associativity* (also called *degree of associativity* or *set size*). The associativity of a cache is the number of block frames in which a given block may reside. Reducing associativity allows fewer block frames to be searched on a reference (a potential implementation advantage), but further constrains which blocks can be simultaneously resident (a potential performance disadvantage).

The terms *fully-associative, set-associative* and *direct-mapped* express the relationship between a cache's associativity and capacity. A cache of c block frames is called *fully-associative* if a block can reside in any block frame (associativity c), *n-way set-associative* if a block can reside only in one of n block frames where $1<n<c$ (associativity n), and *direct-mapped* if a block can reside in only one block frame (associativity 1). Figure 1 illustrates set-associative mapping.

It is worthwhile studying associativity, in particular, because technological and architectural trends are reducing optimal associativity by preferring faster hit times and larger cache sizes. A preference toward faster hit times favors direct-mapped caches, since they have faster hits times than set-associative caches (see Section 5.2). A preference toward larger cache sizes favors direct-mapped caches, since larger cache sizes reduce the miss ratio advantage of set-associative caches (see Section 4.1), allowing cache design decisions to be based on cost and hit time considerations which favor direct-mapped caches (see Sections 5.1 and 5.2).

The technological trends affecting associativity in caches are: (1) faster CPU cycle times relative to memory chip and interconnection speeds, and (2) larger memory chips. The effect of (1) is putting pressure on cache designers to build caches with faster hit times. The effect of (2) is to favor larger caches. For example the decade-old DEC VAX-11/780 has a 200-ns cycles time (in TTL) and an 8K-byte two-way set-associative cache, whereas the recently-introduced DEC VAX 8800 has a 45-ns cycle time (in ECL) and a 64K-byte direct-mapped cache [Clar83, Clar88].

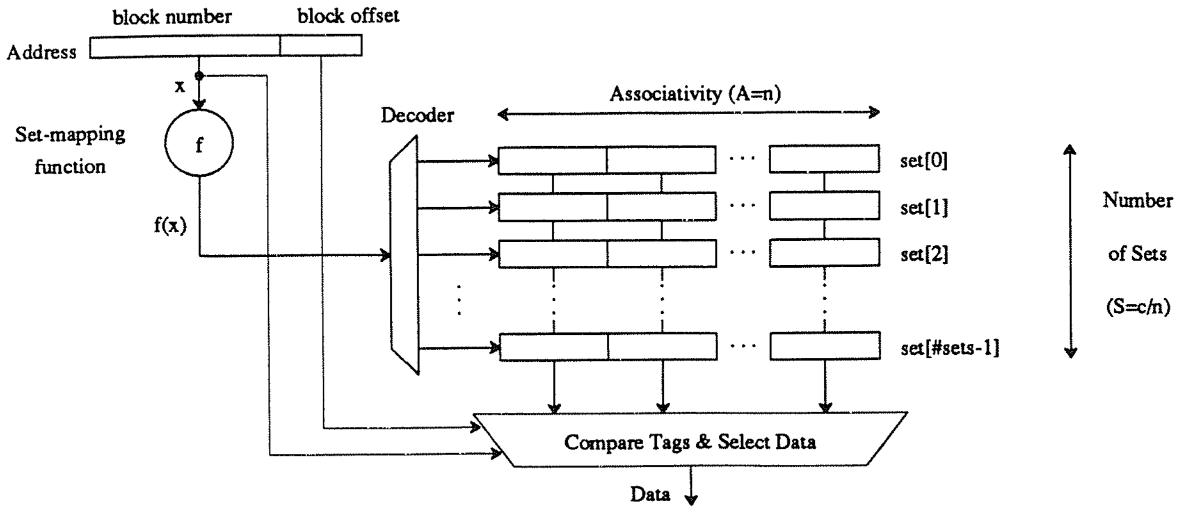


Figure 1. Set-Associative Mapping.

A set-associative cache uses a set-mapping function f to partition all blocks in main memory into a number of equivalence classes. Some number of block frames in the cache are assigned to hold recently-referenced blocks from each equivalence class. Each group of block frames is called a *set*. The number of such groups, equal to the number of equivalence classes, is called the *number of sets*. The number of block frames in each set is called the *associativity* (degree of associativity, set size). The associativity times the number of sets always equals the number of block frames in the cache. A cache is fully-associative if it contains only one set, is direct-mapped if each set contains one block frame, and is set-associative otherwise.

On a reference to block x, set-mapping function f selects one set $f(x)$ (one row), then each block frame is searched until x is found (a cache hit) or the set is exhausted (a cache miss). On a cache miss, one block in set $f(x)$ is replaced with the block x. Finally, the word requested from block x is returned to the processor.

The most-commonly-used set-mapped function is the block number modulo the number of sets, where the number of sets is a power of two. This set-mapping function is called *bit selection* since it equals several low-order bits of the block number.

The architectural trends affecting associativity in caches are the trends toward (1) reduced instruction set computers (RISCs) [Henn84, Patt80, Patt85, Radi82], and (2) shared-memory cache-coherent multiprocessors (multis) [Bell85]†. RISCs exaggerate the need for caches with faster hit times by having simple pipelines that facilitate shorter cycle times, and by referencing memory so frequently (once per cycle) that CPU cycle times are often determined by cache hit times. Machines that reference memory less often, on the other hand, can tolerate multi-cycle caches. More frequent memory referencing also makes RISCs more sensitive to effective (average) access time. A 10 percent decrease in the effective access time of a computer based on the Berkeley RISC II [Kate83] will speed the system by 10

† Commercial RISC processors have been introduced by AMD, Hewlett-Packard, IBM, Intel, MIPS, Motorola, Sun and others. Commercial multis have been introduced by Encore, Sequent, Synapse and others.

percent, whereas a similar decrease in a VAX-11/780-like computer would speed the system by approximately 3.2 percent, since data for the VAX-11/780 suggests it issues 0.32 cache references per cycle [Emer84]. Thus, RISC II can benefit more from an increase in cache size than can a VAX-11/780.

The advent of multis also puts pressure on memory systems to have large caches to reduce processor-memory traffic [Good83]. While using large caches to reduce memory traffic is particularly important for single-bus multiprocessors, it remains important for multiprocessors that use a more-complex processor-memory interconnect [Good88]. The goal of reducing processor-memory traffic may also affect associativity directly by encouraging larger associativity. I do not study this effect in this paper, because of my inability to acquire and analyze memory address traces from functioning multis.

The rest of paper examines associativity in future caches and why I expect direct-mapped caches to be common. Section 2 introduces cache performance metrics. Section 3 illustrates how the implementations of direct-mapped and set-associative caches relate to each other. Section 4 presents arguments against direct-mapped caches and why these arguments will become less important as caches get larger. Section 5 discusses arguments for direct-mapped caches and why these arguments will dominate the arguments against as caches get larger. Finally, Sections 6 and 7 discuss other trends and draw conclusions.

2. Performance Metrics

In this paper I examine cache performance using miss ratio and an extended model of effective access time, but not using other system performance metrics like benchmark execution time or effective number of processors.

Miss ratio is the most commonly-used cache performance metric [Smit82]. The miss ratio for a cache C[†] is:

$$m(C) = \frac{\text{the number of misses with cache } C}{\text{the number of processor memory references}}$$

Miss ratio is used because it is easy to define, interpret, compute [Matt70], and perhaps most important,

† I use $m(C)$, rather than m, to emphasize that the miss ratio is a function of a cache organization. "C" represents all attributes of cache C.

is implementation-independent. This independence facilitates cache performance comparisons between caches not yet implemented and those implemented with different technologies and in different kinds of systems. Unfortunately some comparisons of dissimilar caches can lead to misleading results. A miss ratio comparison, for example, between the Cray-1 instruction buffers and the Motorola 68020 on-chip instruction cache is meaningless because the technologies and workloads have little in common.

Since miss ratio comparisons contrast the number of misses, they can be misleading if the penalty for a miss varies. For instance, increasing cache block size often reduces the number of misses and hence the miss ratio, but it often also increases the number of cycles needed to load a block. The actual change in cache performance will depend on how much the number of misses decreases and on how much the time to service a miss increases [Smit87]. The penalty for a miss can also vary because of delays indirectly affected by changes in miss ratio (e.g., bus interference, writes in a write-through cache, or write-backs in write-back cache).

Effective access time, $t_{eff}(C)$ (average access time), is another commonly-used cache performance metric. Effective access time is the average latency, as seen by the processor, required by the memory system to service a memory reference. In this paper, I model it as:

$$t_{eff}(C) = t_{cache}(C) + m(C)*t_{memory}(C)$$

where $m(C)$, $t_{cache}(C)$ and $t_{memory}(C)$ are the miss ratio, hit time† and average miss penalty (delay beyond a cache access) for cache C.

The principal advantage of using effective access time over using miss ratio is that using effective access time allows caches with different hit and miss times to be more accurately compared. One can, for example, determine whether increasing cache block size improves performance as well as miss ratio. The principal disadvantage of effective access time is that implementation details must be examined and assumptions must be made for the values of $t_{cache}(C)$ and $t_{memory}(C)$. Performance estimates with any implementation assumptions are less general, and those with incorrect assumptions are misleading.

Unlike many other cache memory analyses I do not assume that $t_{cache}(C)$ is the same for all caches studied. The disadvantage of including changes in $t_{cache}(C)$ is that more implementation-dependent

† Strictly speaking, *cache hit time* should be called *cache access time*, since this delay occurs on all accesses, not just hits. I choose not to use *cache access time*, because it is too easily confused with *effective access time*.

parameters must be estimated, further limiting the generality of results. However, as I show in Section 5.3, changes in cache hit time must be included, since ignoring them can lead to incorrect conclusions when comparing large caches of varying associativities. On the other hand, like many other cache memory analyses, I assume that cache changes do not affect the average miss penalty, $t_{memory}(C)$. This assumption simplifies analysis, but can bias results for systems where delays due to write-backs and bus interference are large and variable.

In this paper I do not evaluate caches with system performance metrics like *benchmark execution time* or *effective number of processors,* since these metrics require many system-dependent assumptions which limit their usefulness to comparing similar alternative caches within the context of an existing system. Furthermore, system metrics rarely produce conclusions that generalize to cache designs in other systems, because of the difficulty of isolating cache effects from other system effects.

3. Implementing Caches

This section examines the implementation of direct-mapped and set-associative caches. I concentrate on direct-mapped cache hit (access) logic and *set-associativity* logic, because the delay through this logic determines cache hit time and directly affects effective access time. *Set-associativity logic* is the additional logic required by a set-associative cache over a direct-mapped cache. For this discussion I assume a generic memory system with a single 4G-byte address space of aligned four-byte words, addressed with 32-bit byte-addresses. I assume address translation is done in a way that does not affect the cache hit time.

A direct-mapped cache is simplest to build because the cache location of a referenced word is a function of the address of a reference only and the replacement algorithm is trivial. A direct-mapped cache lookup requires two parallel actions (see Figure 2). One action is to read the word in the cache where the referenced word could reside, and pass it directly back to the CPU. The second action is to read the address tag and state bits for that word to see if the address tag matches and whether the block is valid. A bit is then sent to the CPU indicating a hit if a valid match has occurred, or indicating a miss otherwise. A direct-mapped cache lookup is simpler than a set-associative cache lookup, where reading the data and reading the tags are not independent; instead the tags influence the data selected.

An *n*-way set-associative cache (*n* = 2,4,8 or 16), is a commonly-used cache organization. An *n*-way set-associative cache allows any one of the *n* blocks in the set of a reference to be replaced on a miss. While this flexibility often yields lower miss ratios, it requires that *n* blocks be checked on each reference. To keep a set-associative cache hit time similar to that of a direct-mapped cache, each of the *n* tags in a set must be read and compared to the tag of the reference in parallel. This associative lookup and comparison adds significant cost, as measured in chip count and board area.

Figure 3 shows the basic structure of an *n*-way set-associative cache. Each bank has the same structure as an *n* times smaller direct-mapped cache (see Figure 2). In addition, some logic is needed to select the result from one of the *n* banks. This logic, which I call *set-associativity logic,* can take the form of OR-gates and multiplexors (Figure 3), or wired-ORs and tri-state buffers (Figure 4). The distinction between the logic within the *n* banks and the set-associativity logic is not as clear in many implementations as it is in Figures 3 and 4. For example, the *n* comparators and the encoding logic can be combined into a single *n*-way comparator that directly controls the multiplexor. Nevertheless, a set-associative cache always requires more circuitry than does a direct-mapped cache.

The delay through a set-associative cache is determined by one of three timing paths, illustrated in Figure 5:

(1) *match-found: Address* to *Match[i]* to *MatchOut,*

(2) *select-data: Address* to *Match[i]* to *Select* to *DataOut,* and

(3) *read-data: Address* to *Data[i]* to *DataOut.*

A direct-mapped cache has timing paths *read-data* and *match-found,* but does not have path *select-data,* since the location of cached data in a direct-mapped cache does not depend on which comparator matched.

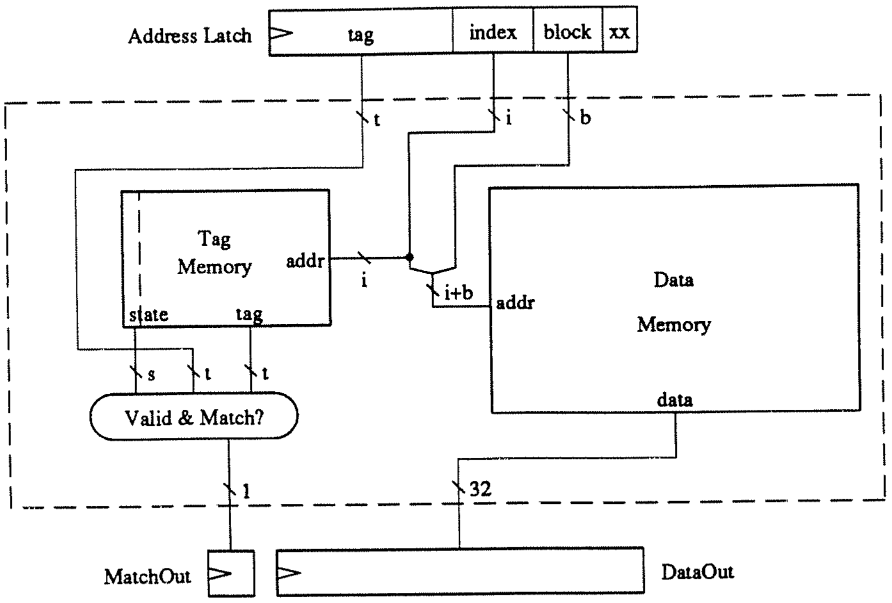


Figure 2. A Direct-mapped Cache.

The figure shows the three components of the access (hit) logic (not the miss logic) for a direct-mapped cache that uses bit selection to select the set (block frame) of a reference. The first component, the *data memory,* holds all the cached data and instructions. Its size is, by definition, the cache size. Conceptually, it is organized as if it were one word wide and accessed with an address formed by concatenating the index and block fields of the address. If it is implemented as a wider memory, some or all of the bits in the block field shift from addressing memory chips to selecting a word after the memory chip access. A block-wide data memory is often preferred when unaligned memory references are permitted.

The second component, the *tag memory,* which holds the state bits (s bits) and address tag (t bits) associated with a cached block, is the cache size in blocks deep and is addressed by the index field. The state bits for a block, usually one or two bits, indicate whether a block is valid (not random bits) and can indicate whether a block is dirty (must be written to main memory on replacement) or its status in a multiprocessor cache coherency protocol. Cache hit logic is only concerned with whether a block is valid.

The last component, the *match logic,* produces a single bit indicating whether the referenced block is present. This bit is asserted only if the tag read from the tag memory is equal to the tag field of the address and the state read from the tag memory is valid.

The address of a reference to a direct-mapped cache is divided into several fields, which from least- to most-significant are: 2 bits that are ignored, since I assume aligned word references; $b = \log_2(\textit{block size in words})$ bits of the block (offset); $i = \log_2(\textit{cache size in blocks})$ bits of the index; and $t = 32 - i - b - 2$ bits of the (address) tag.

A direct-mapped cache lookup requires two parallel actions. One action, called *read-data,* consists of accessing the data memory with the index and block address fields and passing the word read to *DataOut*. The second action, called *match-found* requires two steps. First the tag memory is accessed using the index address field. Second, the address tag and state read from the tag memory are compared by the match logic with the address of the reference to assert *MatchOut* when a cache hit is detected.

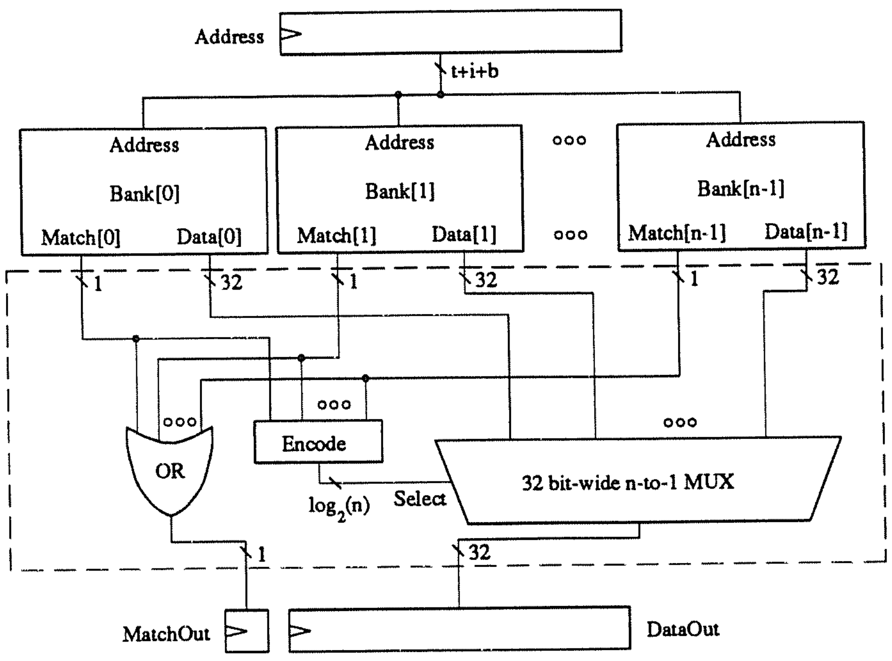


Figure 3. A Set-Associative Cache.

This figure shows cache access (hit) logic for an n-way set-associative cache. This logic consists of n direct-mapped banks and logic to combine the n results into a single match signal and data word. Each bank contains the logic in the dashed box of Figure 2.

Each bank in an n-way set-associative cache of c blocks can be thought of and implemented as a direct-mapped cache of c/n blocks. Therefore the address of a reference is divided as follows: $b = \log_2(\textit{block size in words})$ bits of the block (offset); $i = \log_2(\textit{cache size in blocks} \,/\, n)$ bits of the index; and $t = 32 - i - b - 2$ bits of the (address) tag. On a reference, the address is passed to all the direct-mapped banks. In parallel, each bank selects a block, sends 32-bits of data to *Data[i]*, and computes *Match[i]*, which is asserted on valid tag matches. The *set* of a reference consists of the n blocks selected by the n banks.

After the n direct-mapped caches compute *Match[i]* 's and *Data[i]'* s, the additional logic shown in the dashed box is necessary to produce a single *MatchOut* signal and *DataOut* word. *MatchOut*, asserted on a cache hit, is the logical OR of the n *Match[i]* signals. One way to compute it is with a single n-input OR gate. *Select*, an internal signal, is the number of the bank that matched and can be any value if none matched. *Select* can be computed with an n-bit encoder or with a single level of $\log_2(n)$ $n/2$-input OR gates. *DataOut* must be driven to the value of the bank that matched and can be any value if none matched. One way to compute *DataOut* is with a 32-bit-wide n-to-1 multiplexor (MUX).

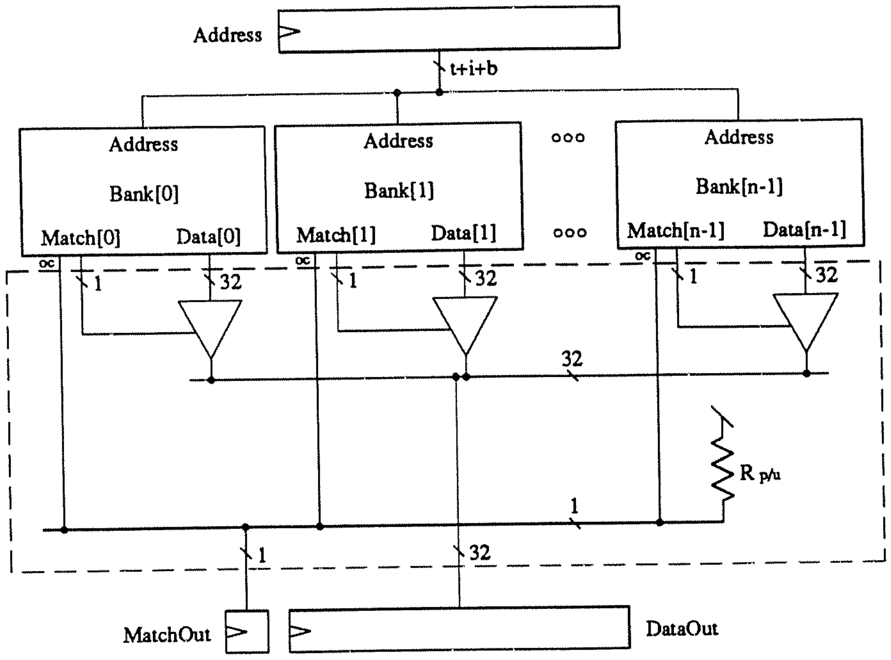


Figure 4. An Alternative Set-Associative Cache.

This figure shows cache hit logic for an n-way set-associative cache using an alternative implementation style. The 32-bit wide n-to-1 multiplexor and select logic have been replaced with n 32-bit-wide tri-state buffers. This alternative does not have to encode *Select*. Instead, each Bank[i] independently enables its tri-state buffer to drive *Data[i]* to *DataOut* if its *Match[i]* is asserted. At most one bank drives *DataOut*, since at most one bank can match. If no banks match, *DataOut* is undefined. The n *Data[i]*'s are connected to each other and *DataOut* with one 32-bit bus. To first order and for small values of n, e.g., $n \leq 8$, the delay to *DataOut* with this design is independent of n.

Alternative logic using wired-OR is also illustrated here for computing *MatchOut*. Each *Match[i]* is computed twice (in parallel) by duplicating the final OR-gate in the match logic (not shown). One copy drives the tri-state enable or multiplexor select, and the other is wired to the other *Match[i]*'s. In TTL, the *Match[i]*'s that are wire-ORed together must be produced with OR-gates using open-collector outputs (''oc''). In ECL, any outputs can be wire-ORed. Two copies of each *Match[i]* are necessary so that the wire-ORing does not affect which data is selected.

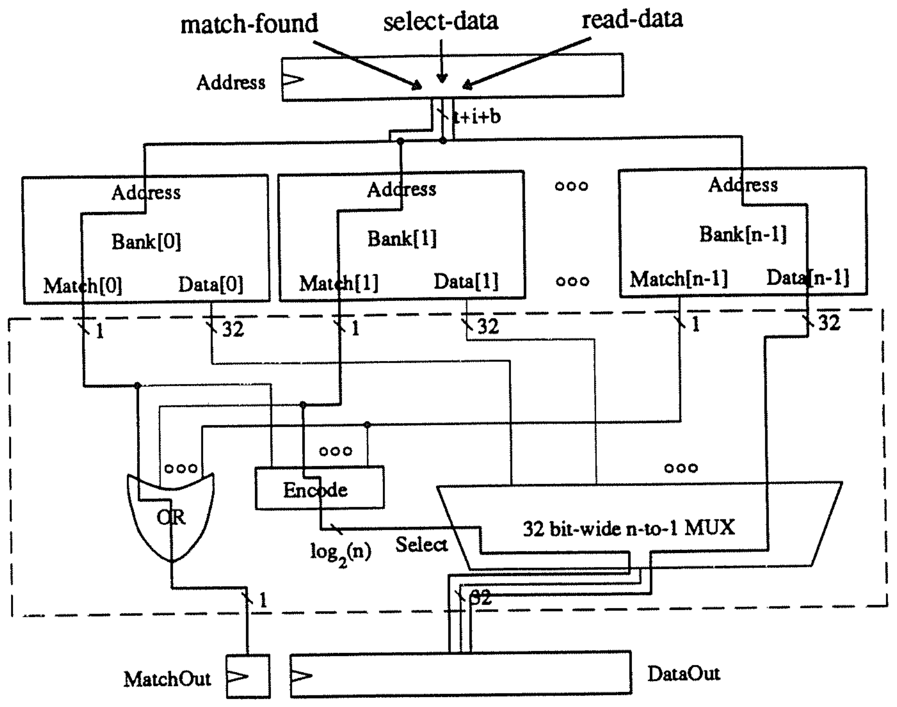


Figure 5. Timing Paths in a Set-Associative Cache.

This figure shows the three timing paths in the cache hit logic for an n-way set-associative cache. The paths are:

(1) *match-found:* signals a cache hit or miss,

(2) *select-data:* selects the data word that corresponds to the tag that matched, and

(3) *read-data:* provides the data on a cache hit.

Path *select-data* is not needed in a direct-mapped cache.

4. Arguments Against Direct-Mapped Caches

The arguments against direct-mapped caches are that: (1) they have worse miss ratios than set-associative caches of the same size, (2) they have terrible worst-case behavior, and (3) they preclude doing address translation in parallel with the first part of the cache lookup. In the following subsections I show that as caches get larger the effects of (1) and (2) are diminished and (3) becomes moot.

4.1. Larger Miss Ratios

It is well-known that direct-mapped caches have larger miss ratios than set-associative caches (see [Matt71], [Kapl73], [Bell74], [Stre76], [Smit78] [Smit82], [Haik84], [Alex86], [Agar87] and [Przy88]). A way to explain this fact is to consider the likelihood of prematurely replacing an active block (one that is being referenced) when multiple active blocks map to the same set. A direct-mapped cache allows only one of the multiple active blocks to reside in the cache at any time, while an n-way set-associative cache allows n blocks to be cached.

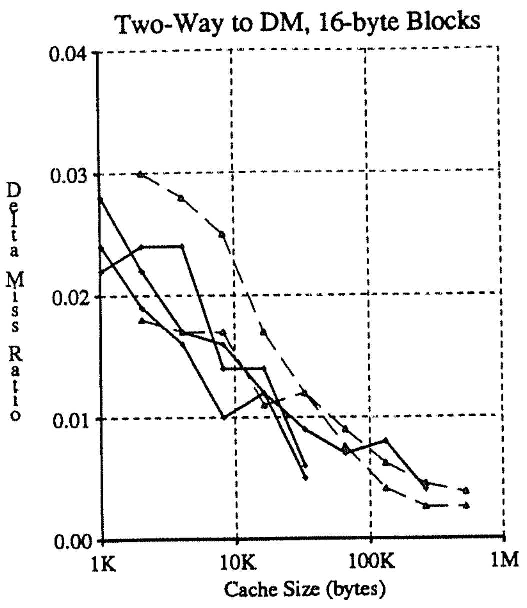
Data from simulation and measurement show, however, that the size of the miss ratio difference that results from changing associativity is less than one might expect (see Figure 6). The intuition that associativity make a tremendous difference is wrong, because it fails to consider that references are not made to random locations. Rather, due to temporal and spatial locality, references are usually made to locations in recently-referenced blocks. The tendency to re-use blocks makes the miss ratios of all caches much less than one, thereby diminishing all potential miss ratio differences.

A trend in the data shown in Figure 6, not heretofore emphasized, is that the miss ratio differences diminish as the caches get larger. For 8K-byte unified (data and instructions cached together) caches with 32-byte blocks, for example, the data show that reducing associativity from two-way to direct-mapped causes an absolute miss ratio change of about 0.013, while at 32K bytes the change is 0.005. The miss ratio differences diminish as the caches get larger for two reasons. First, the active blocks are less likely to map to the same set in larger caches, since larger caches have more sets. For fixed associativity and block size, the number of sets is proportional to cache size. Second, the miss ratios of all cache organizations get smaller with increasing cache size, diminishing potential miss ratio differences.

The data from many sources conclusively shows that the miss ratio difference between a direct-mapped cache and a set-associative cache of the same size diminishes as cache size increases. Consequently, the disadvantage to direct-mapped caches that they have larger miss ratios than set-associative caches becomes less important for larger caches.

4.2. Terrible Worst-Case Behavior

Another argument against direct-mapped caches is that their worst-case behavior, when multiple blocks collide in a set, is terrible. While this is true, one must ask whether an analysis of worst-case



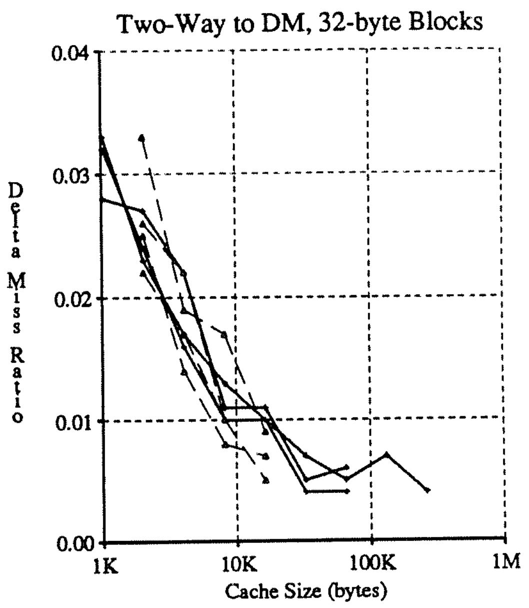


Figure 6. Miss Ratio Differences for Unified Caches.

This figures shows the changes in miss ratio, Δm, which result when associativity is reduced from two-way to direct-mapped for unified (data and instructions cached together) caches with 16-byte (left) or 32-byte (right) blocks. The miss ratio data in both figures (solid lines) is derived from Tables 2 and 3 in [Alex86] and Table 3-4 in [Hill87]. Additional data (dashed lines) for 16-byte blocks (left) comes from Figures 13a and 13b in [Agar87]. Additional data (dashed lines) for 32-byte blocks (right) comes from Figures 10-13 in [Smit82]. Dashed lines should be ignored when comparing 16-byte and 32-byte miss ratios, since this data comes from different traces. The set-associative caches use LRU replacement.

The data show that miss ratio differences diminish as caches get larger.

behavior should include how likely this behavior is. If not, then I submit that the worst-case behavior of direct-mapped caches is no worse than that of set-associative caches. If too many blocks map to a given set, both organizations will "thrash." That fewer active blocks can cause direct-mapped caches to thrash does not change the severity of the worst-case behavior, only its likelihood, which we just chose to ignore.

On the other hand, if one wishes to include the probability that worst-case behavior occurs in one's analysis, then one must observe that (1) worst-case behavior does not occur very often, as is indicated by the small differences in average miss ratios and (2) it occurs less often in larger caches, as is indicated by the diminishing average miss ratio differences.

In summary, the worst-case behavior of all caches, including large caches, is bad, but while worst-case behavior is more likely in large direct-mapped caches than in large set-associative caches, it is still unlikely.

4.3. Parallel Address Translation Difficult

Almost all high-end computers in the last two decades have used paged virtual memory and have organized their caches with physical addresses. In such systems, address translation (the translation of virtual addresses to physical addresses) occurs logically before the cache is accessed. For some such cache configurations, however, it is possible to do the address translation in parallel with part of the cache access. An important disadvantage of reasonably-sized direct-mapped caches is that this technique, called *parallel address translation*, is impractical. In this sub-section I will first explain more about parallel address translation and how it affects cache associativity. Then I argue that as caches get larger, parallel address translation becomes unworkable for all cache organizations, and thus is no longer a disadvantage to direct-mapped caches.

Processors in computers that use paged virtual memory generate memory addresses made up of a (virtual) *page number* and a *page offset*. The physical address, used to address memory, is made up of a (physical) *page frame number* and the same page offset [Denn70]. The memory system must translate page numbers to page frame numbers using mapping information contained in page tables. Since a page may typically reside in any page frame, all address bits except those in the page offset may change during address translation. In a byte-addressable memory system with P-byte pages, the number of bits in the page offset is $\log_2 P$.

Address translation is usually performed using a special-purpose cache of page-number/page-frame-number pairs, called a *translation lookaside buffer* (TLB) or *translation buffer* (TB) [Clar85]. On rare occasions when a page number in not present in the TLB, microcode is usually invoked to access the page tables and load the appropriate entry into the TLB.

The advent of virtual memory presented memory system designers with the decision of whether to access the cache before or after address translation. If the cache is accessed before address translation, i.e., with virtual addresses, then cache tags contain virtual addresses, and the cache is called a *virtually-tagged cache* or just a *virtual cache*. Conversely, a cache accessed after address translation has tags that contain physical addresses and is called a *physically-tagged cache* or a *physical cache*.

The principle disadvantage of a virtually-tagged cache is that hardware may be more complex than for a physically-tagged cache, since virtually-tagged caches must correctly handle situations where two or more virtual addresses that map to the same data block (called *synonyms* or *aliases*) and situations where the same virtual address from two different virtual address space maps to different data blocks. The mapping of physical addresses to data blocks, on the other hand, is one-to-one.

The principal disadvantage of a physically-tagged cache is that an access to it may be slower than that to a virtually-tagged cache, due to the need to do address translation before the cache access. A virtually-tagged cache, on the other hand, only needs to do address translation after a cache access has missed [Wood86].

Until recently, most commercial caches have been physically tagged. One reason for this is that the delay for address translation could usually be hidden using parallel address translation, removing the principal disadvantage of physically-tagged caches.

Parallel address translation hides the delay of address translation by doing the TLB access in parallel with reading out the physical-address tags and data associated with each block frame in the set of the reference. The set is selected using index bits from the *physical* address. The only straightforward way to know the value of the index bits before the TLB access is to require that they come from the page offset, which is not altered by address translation. The index bits are contained in the page offset if the number of sets in the cache does not exceed the number of blocks in a page or:

$$cache\_size \leq associativity * page\_size.$$

In a system with 4K-byte pages, for example, this invariant limits a straight-forward four-way set-associative cache to 16K bytes and a direct-mapped cache to an unreasonable 4K bytes.

For existing architectures, with fixed page sizes, parallel address translation requires that each doubling of cache size be accompanied by a doubling of associativity. The IBM 3033, for example, has 16-way set-associative, 64K-byte, physically-tagged cache and 4K-byte pages. Clearly its associativity was not selected from miss ratio considerations, since 16-way and eight-way set-associative miss ratios are almost identical and a 16-way set-associative cache is more expensive to implement†.

† The cost of such a cache can be reduced by building a 16-way set-associative tag memory (directory) together with a data memory of more limited associativity (e.g., four-way), as is done in recent IBM mainframes. In such a 64K-byte cache, a set of 16 address tags is selected with bits from the page offset, and a set of four data blocks is selected with the page offset bits and two bits from the page number. Now three cases are possible: a normal cache hit (tag match and data selected), a normal cache miss (no tag match), or a cache

Parallel address translation will become impractical in existing architectures as caches get larger. Eventually the increased hit time and implementation costs of wider associativity will overwhelm the benefits of parallel address translation.

In new systems, the life of parallel address translation can be extended by using large page sizes and/or restricting the mapping of pages to page frames so that some low-order bits in the page number do not change in address translation. Allowing larger pages (e.g., 64K bytes as in a research machine at DEC WRL [Joup86]), permits parallel address translation in caches of 64K, 128K and 256K bytes if they have associativities of 1-, 2- and 4-way. However, I expect subsequent implementations of new architectures to use even larger caches. These caches will require wider associativities since few architectures permit variable-sized pages. Thus using larger page sizes, only defers the point at which parallel address translation requires unreasonably wide associativity.

Restricting the mapping of pages to page frames can provide additional invariant bits for indexing the cache. Requiring, for example, that the three low-order bits of a (virtual) page number be the same as the three lower-order bits of the (physical) page frame number provides three more bits for indexing the cache. This approach has three problems, which from most to least important are: (1) handling two arbitrary pages that map to the same page frame, (2) having the virtual memory software manage eight, rather than one, pool of pages and page frames, and (3) suffering a higher page fault rate, since physical memory is divided into eight sets instead of one. While solutions exist for each of these problems, I assert that the complexity cost incurred is too high to pay for the benefit of retaining parallel address translation.

Thus, parallel address translation will become impractical in existing and new architectures as caches get much larger than the page size. Eventually the increased hit time and implementation costs of wider associativity will overwhelm the benefits of parallel address translation, leaving designers with the choice of doing address translation before an access to a physically-tagged cache or after an access to a virtually-tagged cache. In either case, the disadvantage to direct-mapped caches that they prevent parallel address translation while other set-associative caches permit it is removed.

hit on data loaded under an alias (tag match and data not selected). Since one expects the final case to be rare, an acceptable way handling it is to move the data block to one of the selected block frames, replacing data that was there before. This cache has the miss ratio and data memory cost of a four-way set-associative cache plus the cost of a 16-way tag memory (including 16 comparators) and some additional control complexity.

5. Arguments For Direct-Mapped Caches

The arguments for direct-mapped caches are that: (1) they can be implemented at less cost than set-associative caches, (2) their cache hit (access) times are smaller than those of comparable set-associative caches, and (3) they have smaller effective (average) access times than set-associative caches for sufficiently large cache sizes. In the following sub-sections I support the above arguments, and show why I expect a direct-mapped organization to be commonly-used for large caches (e.g., ≥ 64K bytes).

5.1. Lower Cost

A direct-mapped cache never costs more than a set-associative cache, because there exists a way to convert a set-associative design into a direct-mapped one at no cost†. (The cost of a cache can be measured in many dimensions, such as number of chips, chip area, power-consumption, dollars and design time.) An n-way set-associative cache, like the one shown in Figure 3, can be converted to a direct-mapped one simply by changing the replacement algorithm. On a cache miss, an n-way set-associative cache selects a victim, a block to be replaced, using some algorithm (e.g., LRU or random). A direct-mapped cache is created if the victim is selected with the lower $\log_2 n$ bits of the address tag of the new reference. Since this replacement algorithm requires less hardware than the original replacement algorithm, a direct-mapped cache will cost less than a set-associative one.

In practice, direct-mapped caches cost significantly less, since less parallelism is required. An n-way set-associative cache must read n tags in parallel and compare each of them with the high-order bits of the reference's address. A direct-mapped cache need only read and compare one tag. Thus direct-mapped caches need fewer comparators, require fewer connections and can use fewer, larger (deeper) memory chips. Similarly, the data memory (and connections to it) in an n-way set-associative cache must be n times as wide as that for a direct-mapped cache, enabling the direct-mapped cache to use fewer, larger memory chips.

† Assuming parallel address translation is not being done.

5.2. Faster Hit Time

The hit (access) time of a direct-mapped cache is less than or equal to that of a comparable set-associative cache. It is at least equal, because the transformation described above creates a direct-mapped cache with exactly the same hit time as a set-associative cache.

In practice, the hit time of a direct-mapped cache is less than that of a comparable set-associative cache, because the critical timing path can be made shorter. The delay paths, displayed in Figure 5, are:

(1) *match-found:* signals a cache hit or miss,

(2) *select-data:* selects the data word that corresponds to the tag that matched, and

(3) *read-data:* provides the data on a cache hit.

The hit time of a direct-mapped cache can be less than that of a set-associative cache, because the *select-data* path can be eliminated in a direct-mapped cache. Instead of letting the results of tag comparisons determine the data returned to the CPU, the data can be selected with several bits from a reference's addresses. These bits can directly control a multiplexer or be decoded to control tri-state buffers. In either case, this timing path is so much faster than the others that it is effectively eliminated. Figure 7 illustrates this improvement.

An important effect of eliminating the *select-data* timing path is that the *match-found* and *read-data* paths are now independent. This makes it possible for a direct-mapped cache to return the correct data and the CPU to resume execution even before the system knows whether a hit will occur, so long as the CPU can back out of execution begun with incorrect data. This optimistic use of cache data is being used in a research machine [Dion86], where it enables the cache hit time and the machine cycle time to be reduced by approximately one-third. Optimistic use of cache data is possible in a set-associative cache if one always returns the most-recently-used (MRU) block in the selected set [Chan87]. I found, however, that the performance of a simple direct-mapped cache is similar to that of a more-complex MRU cache [Hill87].

It is also possible to improve the *read-data* path, since it is no longer necessary to read from n data blocks in parallel. Instead only one block need be read. This flexibility allows designers to organize data memory chips differently and to use larger, deeper chips. It is possible, for example, to completely eliminate the multiplexer or tri-states previously used to select data from different blocks.

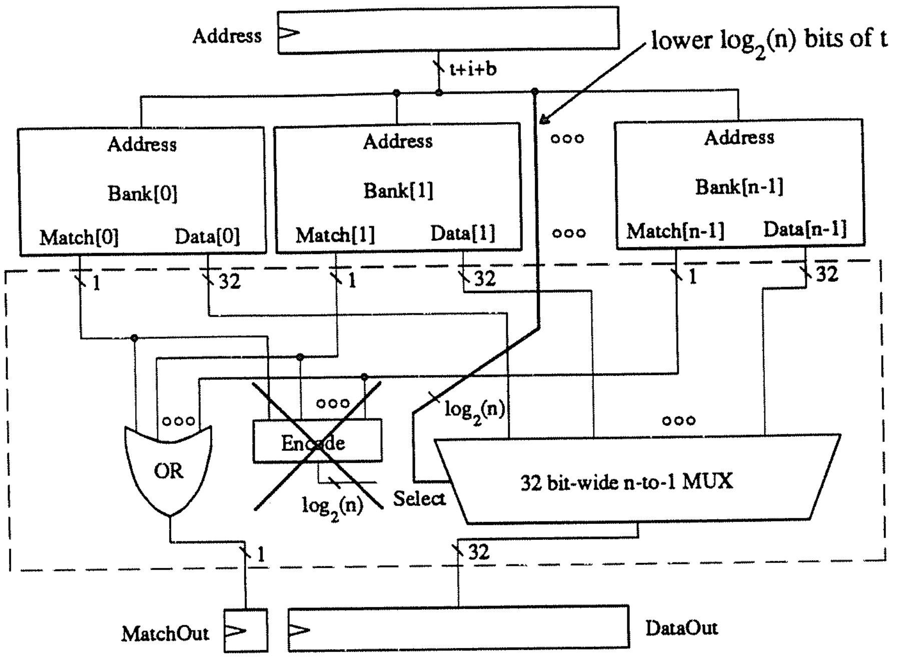


Figure 7. Converting a Set-Associative Cache into a Direct-Mapped One.

An n-way set-associative cache can be converted into a direct-mapped cache by changing the replacement to replace the block in bank r, where r is the reference's tag modulo n. Since this function can be done with bit selection (at trivial cost) and off the critical path for a cache hit, the resulting direct-mapped cache has the same cost and hit time as the original set-associative cache. Thus, moving to a direct-mapped cache never requires that cost or hit time be increased. On the other hand, the text describes several ways moving to a direct-mapped cache decreases cost and hit time.

This figure illustrates how the hit time of a direct-mapped cache will be less than that of a set-associative cache if the critical timing path for the set-associative cache is *select-data*. This path can be eliminated in a direct-mapped cache in favor of selecting the data with the least-significant $\log_2 n$ bits from the tag of the set-associative design.

Finally, improvement in the *match-found* path is also possible, since it is no longer necessary to read and compare n tags in parallel and to then "OR" the results for the cache hit/miss signal. Rather one need only read and compare one tag. This flexibility allows the tag memory to be implemented with fewer, deeper chips and eliminates the final "OR" stage.

The exact magnitude of the improvement possible depends on many implementation factors. In my thesis [Hill87], I examined caches implemented in three technologies: (1) TTL logic and MOS SRAM memory chips, (2) ECL logic and memory chips, and (3) custom CMOS. I found that moving from a direct-mapped to a two-way set-associative cache increases cache hit time in (1) from 100 to 109 ns (9%), in (2) from 30.0 to 33.5 ns (12%), and in (3) from 50.0 to 51.0 ns (2%).

A fair summary of these numbers is that the difference is about 10 percent for board-level TTL and ECL caches and much smaller for custom CMOS caches. I do not regard the difference the between the TTL and ECL times as significant, since both numbers are very sensitive to the propagation delays through a few parts. Since custom CMOS assumptions are radically different from those for MSI, comparing CMOS results with TTL or ECL results is subject to more error. However, one should expect the penalty for adding a multiplexer to be larger in MSI, where it adds logic delay and two chip crossings, than on a custom chip, where it adds just the logic delay.

In summary, the hit time of a direct-mapped cache will be less than that of a comparable set-associative cache, since block selection can be done before the tag comparison completes, and the tag and data memories do not need to read information from n blocks in parallel.

5.3. Superior Effective Access Times

A direct-mapped cache has a smaller effective (average) access time than that of a set-associative cache of the same size if (1) the direct-mapped cache has a smaller hit time and (2) both caches are sufficiently large that the miss ratio difference between them is small. Furthermore, direct-mapped caches can have superior effective access times at practical cache sizes (e.g., 64K bytes for unified and data caches, or 16K bytes for instruction caches).

Effective access time, $t_{eff}(C)$, is the average latency, as seen by the processor, required by the memory system to service a memory reference. As discussed in Section 2, I model $t_{eff}(C)$ as:

$$t_{eff}(C) = t_{cache}(C) + m(C)^{*}t_{memory}(C)$$

where $m(C)$, $t_{cache}(C)$ and $t_{memory}(C)$ are the miss ratio, hit time (cache access time) and average miss penalty for cache C.

If two caches have the same miss penalty, the change in effective access time moving from a cache C_1 to a cache C_2 is:

$$\Delta t_{eff} = \Delta t_{cache} + \Delta m^{*} t_{memory}$$

where:

$$\Delta t_{eff} = t_{eff}(C_2) - t_{eff}(C_1),$$

$$\Delta t_{cache} = t_{cache}(C_2) - t_{cache}(C_1),$$

$$\Delta m = m(C_2) - m(C_1), \text{ and}$$

$$t_{memory} = t_{memory}(C_1) = t_{memory}(C_2).$$

If cache C_1 is direct-mapped and cache C_2 set-associative, then $\Delta t_{cache} \geq 0$ and $\Delta m^* t_{memory} \leq 0$, since set-associative caches typically have a slower hit time and smaller miss ratio than direct-mapped caches of the same size. Figure 8 illustrates the Δt_{eff} for hypothetical direct-mapped and set-associative caches. It shows that Δt_{eff} can be either positive or negative. If on the other hand implementation considerations are ignored, then:

$$\Delta t_{eff} = 0 + \Delta m^* t_{memory} \leq 0,$$

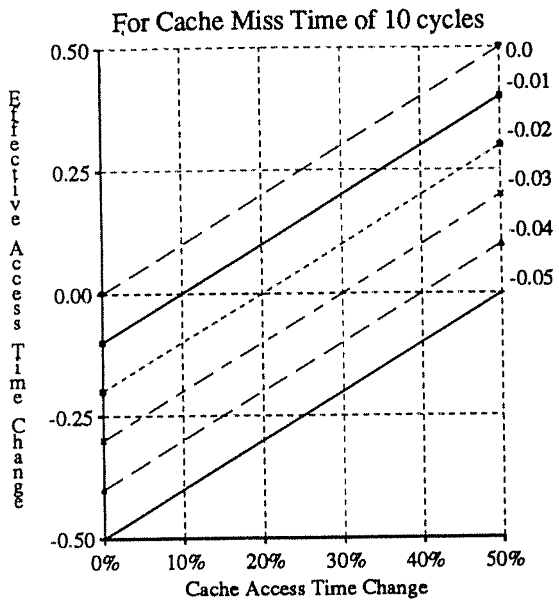
which implies increasing associativity always improves effective access time (Δt_{eff} is negative). Thus, the effect of including implementation considerations is to diminish or reverse the miss ratio benefit of increasing associativity.

To see whether implementation considerations matter in practice, I must find typical values for t_{memory}, Δm and Δt_{cache}. Reasonable values for t_{memory} are 10 or 20. Smaller values are possible, especially in systems where cache misses are serviced by larger caches, instead of main memory. Larger values are possible in a system where the mismatch between the technologies used to implement the cache and memory is larger than normal.

Typical values for Δm, the absolute difference in miss ratio, can be derived from trace-driven simulation. Figure 9 shows miss ratio differences between some direct-mapped and two-way set-associative caches with 32-byte blocks. The data show that Δm's generally get smaller as cache size is increased, and that the absolute values of the Δm's are small for larger caches†. All Δm's for caches larger than 16K bytes, for example, are less than 0.01.

Figure 10 shows effective access time changes with actual miss ratio differences for unified caches from Figure 9. Lines are labeled the cache sizes and positioned based on the miss ratio difference for that cache size. Figures 11 and 12 show similar results for instruction and data caches. These figures illustrate three points regarding moving from a direct-mapped to a two-way set-associative cache:

† Results in [Hill87] show that miss ratio differences are roughly proportional to absolute miss ratios. For instance, decreasing associativity from two-way to direct-mapped in unified caches, causes a relative miss ratio increase of about 30 percent.



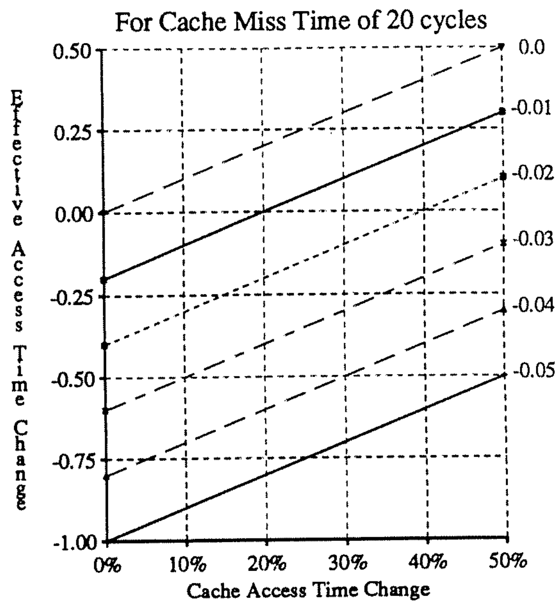


Figure 8. Change in Effective Access Time.

This figure shows the change in effective access time that results from moving from a cache with a relatively fast hit time and a relatively large miss ratio (e.g., a direct-mapped cache) to another cache with a slower hit time but smaller miss ratio (e.g., a set-associative cache). The graphs assume 10- (left) and 20-cycle (right) miss penalties, where a cycle is defined to be equal to the hit time of the faster cache. The x-axis displays values of Δt_{cache}, the hit time difference. An x value of 20% implies that the slower cache's hit time is 1.2 cycles, 1.2 times the hit time of faster cache. The y-axis gives values of Δt_{eff}, the change in effective access time. A y value of -0.10 implies that the effective access time improves by 0.10. Since most effective access times are slightly larger than 1.0, an absolute improvement of 0.10 cycles translates into slightly less than a 10 percent relative improvement. The various lines show miss ratio changes, Δm, from -0.05 up to 0.0. All Δm's are non-positive, since we assume the second cache has a smaller miss ratio.

Points on the y-axis represent the effective access time change that results when Δt_{cache} is zero or ignored. Here all points are below the x-axis, since the latter cache, with the smaller miss ratio, always has a better effective access time ($\Delta t_{eff} < 0$).

If $\Delta t_{cache} > 0$, the benefit of the lower miss ratio is diminished. For all points above the x-axis, the drawback of the slower hit time exceeds the benefit of the lower miss ratio, making the former cache preferred ($\Delta t_{eff} > 0$).

(1) Moving from a direct-mapped to a two-way set-associative cache has little potential for improving effective access time as caches get larger. At 64K bytes (see lines labeled *64K*) and with 10-cycle misses, the maximum improvement possible is 5.2, 3.6 and 4.5 percent for unified, instruction and data caches. With 20-cycle misses, the maximum possible improvement is twice as large.

(2) Moving from a direct-mapped to a two-way set-associative cache can cause a worse effective access time if cache hit time increases by even a small amount. The improvement is offset if the cache time increase is equal to the maximum improvement possible from the smaller miss ratio (e.g., 5.2, 3.6 and 4.5 percent for unified, instruction and data caches of 64K bytes, having 10-

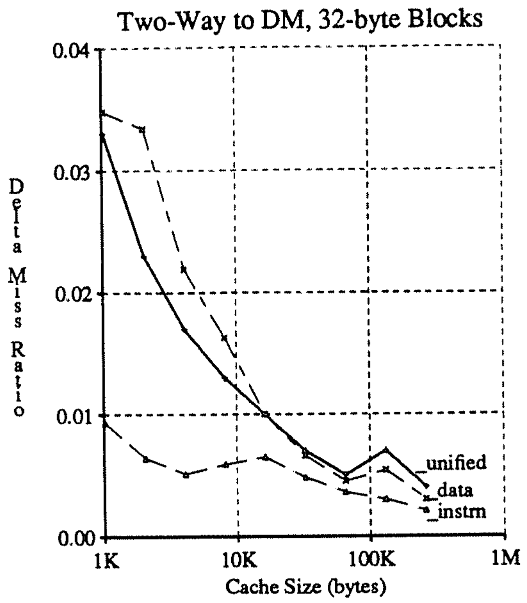


Figure 9. Miss Ratio Differences.

This figure displays the miss ratios from a direct-mapped caches less the miss ratio of two-way set-associative caches of the same size for unified, instruction and data caches with 32-byte blocks from Table 3-4 in [Hill87]. Results show miss ratio differences generally diminish with increasing cache size, and are smaller for instruction caches than for unified or data caches.

cycle miss penalties).

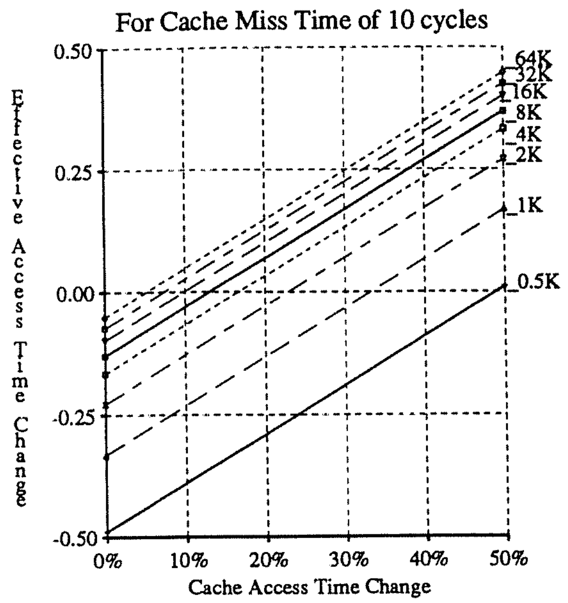
(3) Moving from a direct-mapped to a two-way set-associative cache offers less to instruction caches than it does to unified or data caches. The potential benefit from increasing associativity in instruction caches with a 10-cycle miss time is less than 6.4 percent for sizes as small as 2K bytes. The actual benefit will be less if the miss penalty is less than 10 cycles or increasing associativity impacts cache hit time.

The final parameter value that must be determined to know whether direct-mapped or set-associative caches are faster is Δt_{cache}. This parameter is difficult to determine, because it is implementation-dependent and very sensitive to the delay through a few parts.

As discussed in the previous sub-section, I examined some board-level caches (TTL and ECL) where Δt_{cache} was around 10 percent. The effect of a ten percent slow down can be studied in Figures 10, 11 and 12 by only considering design points on a vertical line at $\Delta t_{cache} = 10\%$. For the 10-cycle miss penalty, $\Delta t_{cache} = 10\%$ implies that direct-mapped caches have better effective access times than two-way set-associative caches for caches equal to and larger than 16K, 8K and 16K bytes for unified, instruction and data caches. For the 20-cycle miss penalty, the corresponding sizes are 64K, 16K and 64K bytes.

The exact cache size at which the effective access time of a direct-mapped cache becomes better than that of a two-way set-associative cache is sensitive to many assumptions. Nevertheless, that it does *crossover* is inevitable, given that miss ratio differences diminish as caches get larger and that set-associative caches have slower hit times.

At cache sizes less than the crossover size, a direct-mapped cache may still be preferred to a set-associative one, since a direct-mapped cache may cost less and its effective access time may not be much worse. Even for 20-cycle miss penalties, an examination of Figures 10, 11 and 12 shows that the effective access time of a two-way set-associative cache is never more than five percent better than that of the corresponding direct-mapped cache at cache sizes 32K bytes and larger.



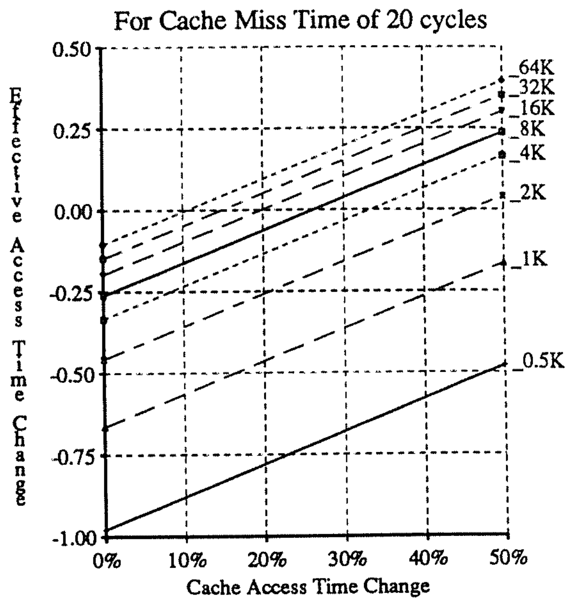
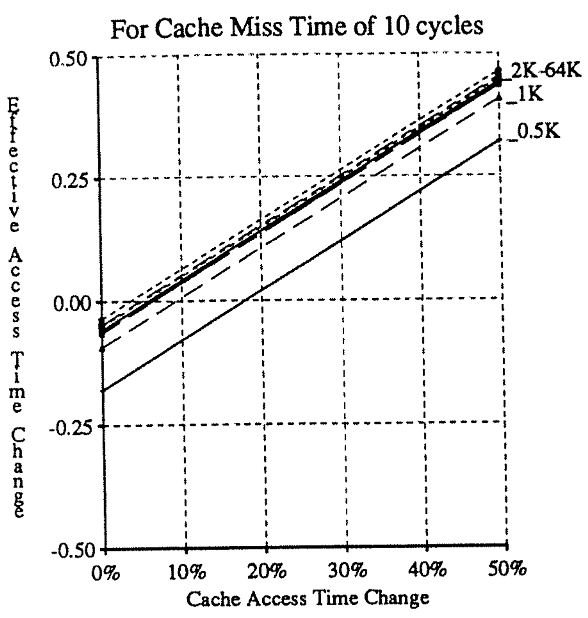


Figure 10. Effective Access Time Changes in Unified Caches.

This figure shows the change in effective access time that results from moving from a direct-mapped cache to a two-way set-associative cache when both caches are unified, have 32-byte blocks and have 10- (left) or 20-cycle (right) miss penalties. This figure is constructed by substituting miss ratio differences for unified caches from Figure 9 into Figure 8. The lines are labeled with cache sizes in bytes and positioned by the miss ratio difference at that cache size.

The data for 16K-byte caches with 10-cycle miss penalties, for example, can be interpreted as follows: increasing associativity from direct-mapped to two-way improves effective access time by 0.10 if there is no speed cost to adding associativity ($\Delta t_{cache} = 0$); increasing associativity has no effect on effective access time if the set-associative cache's hit time is 10 percent longer; and increasing associativity causes a worse effective access time, despite lowering the miss ratio, if the set-associative cache more than 10 percent slower.



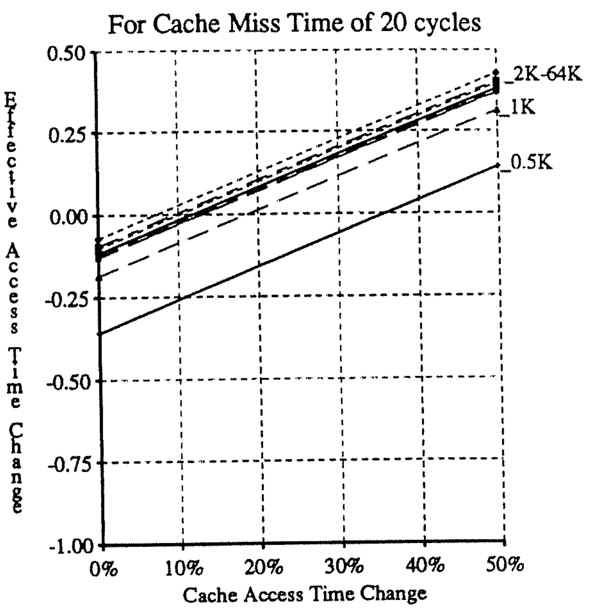
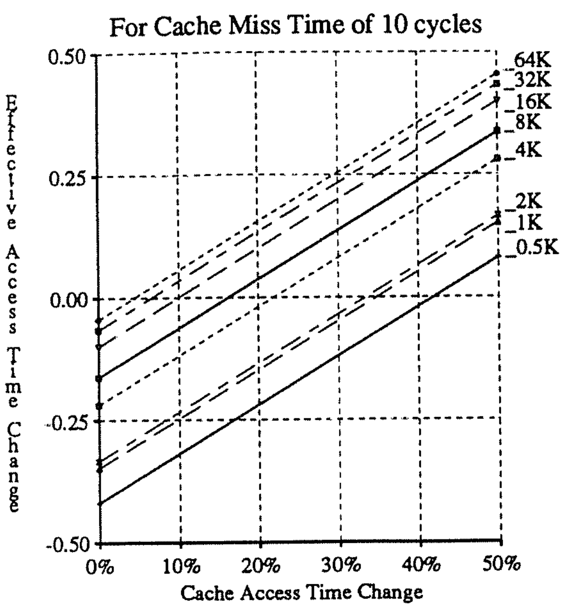


Figure 11. Effective Access Time Differences in Instruction Caches.

This figure shows the effective access time changes that result from moving from a direct-mapped instruction cache to a two-way set-associative instruction cache with miss penalties of either 10 (left) or 20 (right) cycles. Other assumptions match those of Figure 10.

The benefit of associativity is smaller for instruction caches than it is for unified or data caches.



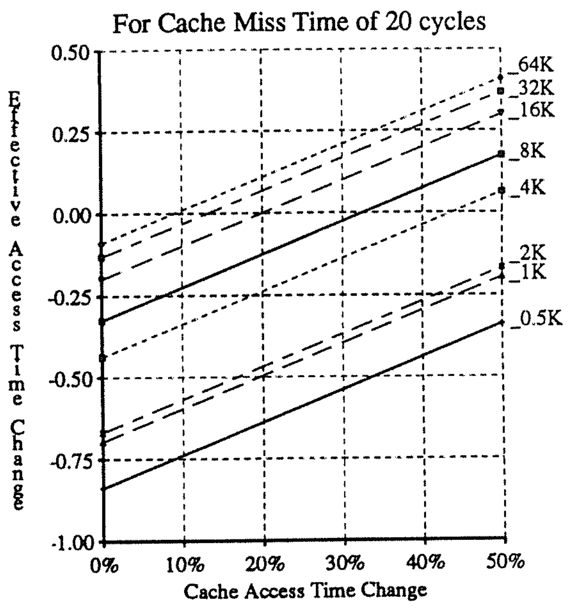


Figure 12. Effective Access Time Differences in Data Caches.

This figure shows the effective access time changes that result from moving from a direct-mapped data cache to a two-way set-associative data cache with miss penalties of either 10 (left) or 20 (right) cycles. Other assumptions match those of Figure 10.

The benefit of associativity in data caches is similar to that for unified caches.

6. Other Trends

In addition to expecting direct-mapped caches to be common in future systems, I also see trends toward virtually-tagged caches and hierarchies of caches. I foresee the use of virtually-tagged caches in systems that require fast caches with low miss ratios. Physically-tagged caches are more common today, because they are simpler (no need to worry about synonyms or multiple address spaces) and the delay of address translation can often be hidden using parallel address translation. Many future systems will need fast caches, because cache hit time will often determine the cycle time and/or pipeline length, particularly in RISC processors. The desire for lower miss ratios will force larger caches sizes, making parallel address translation impossible. The benefit of not having to do address translation before every cache access will justify the complexity of handling synonyms, making virtually-tagged caches worthwhile. The cost of handling synonyms is not as great as previously supposed, especially in cache-coherent multiprocessors [Good87].

For several reasons, I also expect cache hierarchies, heretofore rarely-used, to become more common in future systems as processors speed up relative to main memories. First, implementation considerations can force a partition. Some recently-introduced microprocessors, for example, devote some of their limited on-chip area to caches, but require larger caches to avoid frequent accesses to relatively slow main memory. Since the on-chip caches cannot be made larger, a second on-board cache is required. Second, a detailed computation of effective access time shows that multiple levels of caches can offer superior performance to a single cache [Przy88]. Third, there may be functional and performance benefits to specializing caches at different levels. A first-level cache (nearest to the CPU) can be optimized to minimize effective access time while the last-level cache is designed to reduce cost or interconnection traffic. Similar reasons are given in [Shor88].

The utility of direct-mapped caches in cache hierarchies is, as yet, undetermined. First-level caches will be direct-mapped if technological constraints permit large enough cache sizes that the hit time advantage of direct-mapped caches (due in part to allowing data to be returned before the tag comparison is complete) is more important than the miss ratio disadvantage. Second-level caches will be direct-mapped if lower cost is more important than minimizing processor-memory traffic.

7. Conclusions

Direct-mapped caches will be important in many future computer systems where caches are large ($\geq$ 64K bytes), since the principal arguments against direct-mapped caches with respect to set-associative caches diminish with increasing cache size and the arguments for direct-mapped caches do not.

The arguments against direct-mapped caches, with respect to set-associative caches, are that they have (1) worse miss ratios, (2) more-common worst-case behavior and (3) preclude parallel address translation. I have shown that the significance of (1) and (2) becomes questionable for large caches where absolute miss ratio differences are small, and that (3) is not a disadvantage for large direct-mapped caches, since large set-associative caches also preclude parallel address translation.

The arguments for direct-mapped caches are that they (1) cost less, (2) have faster hit (access) times, and (3) can have superior effective (average) access times. I have shown that the strength of these arguments is not diminished by increasing cache size, and that (3) is more likely to be true for large cache sizes. Furthermore, using some implementation assumptions, I show that some practical direct-mapped caches have better effective access times than two-way set-associative caches at 64K bytes and larger for unified and data caches and 16K bytes and larger for instruction caches.

An alternate way of stating the above result is:

- set-associative caches reduce the time spent on cache misses,
- direct-mapped caches reduce the time spent on cache hits,
- set-associative caches are preferred in small caches where misses are common,
- direct-mapped caches are preferred in large caches where misses are rare,
- many future caches will be large, and therefore direct-mapped.

8. Acknowledgements

I would like to thank my thesis advisors, Alan Smith and David Patterson, for their many suggestions that improved the quality of my research. Thanks also to those who read and improved drafts of this paper: Sue Dentinger, James Goodman, David Patterson and Gurindar Sohi.

The material presented here is based on research supported in part by the Defense Advanced Research Projects Agency monitored by Naval Electronics Systems Command under Contract No. N00039-85-C-0269, the National Science Foundation under grants CCR-8202591 and MIP-8713274, by the State of California under the MICRO program and by IBM Corporation, Digital Equipment Corporation, Hewlett Packard Corporation, and Signetics Corporation.

9. References

[Agar87] A. Agarwal, M. Horowitz and J. Hennessy, Cache Performance of Operating Systems and Multiprogramming Workloads, submitted to TOCS (April 1987).

[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.

[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.

[Bell85] C. G. Bell, Multis: A New Class of Multiprocessor Computers, *Science,* 228 (26 April 1985).

[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.

[Clar85] D. W. Clark and J. S. Emer, Perfomance of the VAX-11/780 Translation Buffer: Simulation and Measurement, *ACM Trans. on Computer Systems,* 3, 1 (February 1985).

[Clar88] D. W. Clark, P. J. Bannon and J. B. Keller, Measuring VAX 8800 Performance with a Histogram Hardware Monitor, *15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii (June 1988).

[Denn70] P. J. Denning, Virtual Memory, *Computing Surveys,* 2, 3 (September 1970).

[Dion86] J. Dion, *Private Communication*, Dec.. Western Research Lab, (May 1986).

[Emer84] J. S. Emer and D. W. Clark, A Characterization of Processor Performance in the VAX-11/780, *Proc. Eleventh International Symposium on Computer Architecture*, Ann Arbor, MI (June 1984).

[Good83] J. R. Goodman, Using Cache Memory to Reduce Processor-Memory Traffic, *Proc. Tenth International Symposium on Computer Architecture*, Stockholm, Sweden (June 1983), 124-131.

[Good87] J. R. Goodman, Coherency for Multiprocessor Virtual Address Caches, *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems,* Palo Alto, California (October 1987).

[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).

[Haik84] I. J. Haikala and P. H. Kutvonen, Split Cache Organizations, CS Report C-1984-40., Univ. of Helsinki (August 1984).

[Henn84] J. L. Hennessy, VLSI Processor Architecture, *IEEE Trans. on Computers,* C-33, 12 (December 1984).

[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).

[Joup86] N. Jouppi, *Private Communication*, Dec. Western Research Lab, (December 1986).

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

[Kate83] M. G. H. Katevenis, R. W. Sherburne, D. A. Patterson and C. H. Séquin, The RISC II Micro-Architecture, *Proc. VLSI 83 Conference*, Trondheim, Norway (August 1983).

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

[Matt71] R. L. Mattson, Evaluation of Multilevel Memories, *IEEE Trans. on Magnetics*, MAG-7, 4 (December 1971).

[Patt80] D. A. Patterson and D. R. Ditzel, The Case for the Reduced Instruction Set Computer, *Computer Architecture News*, 8, 6 (15 October 1980), 25-33.

[Patt85] D. A. Patterson, Reduced Instruction Set Computers, *Comm. ACM* (January 1985), 8-21.

[Przy88] S. Przybylski, M. Horowitz and J. Hennessy, Performance Tradeoffs in Cache Design, *15th Annual International Symposium on Computer Architecture*, Honolulu, Hawaii (June 1988).

[Radi82] G. Radin, The 801 Minicomputer, *Proc. Symposium on Architectural Support for Programming Languages and Operating Systems*, Palo Alto, California (March 1-3, 1982), 39-47.

[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.

[Smit87] A. J. Smith, Line (Block) Size Choice for CPU Caches, *IEEE Trans. on Computers*, C-36, 9 (September 1987).

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

[Wood86] D. A. Wood, S. J. Eggers, G. Gibson, M. D. Hill, J. Pendleton, S. A. Ritchie, R. H. Katz and D. A. Patterson, An In-Cache Address Translation Mechanism, *Proc. Thirteenth International Symposium on Computer Architecture*, Tokyo, Japan (June 1986).