A Case for Direct-Mapped Caches

Mark D. Hill
University of Wisconsin

Despite having worse miss ratios, large direct-mapped caches often handle processor references faster than more-expensive set-associative caches.

A cache is a small, fast buffer in which a system keeps those parts of the contents of a larger, slower memory that are likely to 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 caches are successful due to temporal and spatial locality, two properties of most real reference streams. Temporal locality means future references are likely to be made to the same locations as recent references, 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.[1] Like caches in general, CPU caches are faster and smaller than the memory they buffer. They are usually five to 20 times faster and 50 to 1,000 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.

CPU caches have been studied extensively[2] because they 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 technological improvements are altering their characteristics. (Since this article examines only CPU caches, the term cache is often used instead of CPU cache.)

The important cache design parameter examined here is associativity, which is 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. However, this 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 because technological trends toward large, fast static RAMs are facilitating larger cache sizes and architectural trends toward reduced instruction set computers (RISCs) are requiring faster hit times. The trend to larger caches is illustrated by the VAX 11 family. The recently introduced VAX 8800 uses a 64-Kbyte direct-mapped cache, while older VAX 11 implementations like the VAX 11/780 and VAX 11/785 use set-associative caches of 8 and 16 Kbytes.

RISCs accentuate the need for caches

0018-9162/88/1200-0025$01.00 © 1988 IEEE

Selected CPU-cache terminology

Basic CPU-cache terms used throughout this article are defined below. Some of these terms are also illustrated in Figure 1.

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.)

memory A larger, slower memory that provides data not found in the cache. (Synonyms: main memory, primary storage.)

reference A request by the processor to read or write a memory location. (Synonyms: request, access, processor reference, memory reference.)

hit, miss References found in the cache are said to hit; those not found, to miss.

miss ratio A cache performance metric giving the probability that a reference misses.

effective access time A cache performance metric giving the average time required to service a reference. (Synonym: average access time.)

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 is valid 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. (Synonyms: block, line.)

block Data from memory that fills a block frame. (Synonym: line.)

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.

with fast 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. Commercial RISC processors have been introduced by AMD, Hewlett-Packard, IBM, Intel, MIPS, Motorola, Sun, and others.

This article will show that trends toward larger cache sizes and faster hit times favor direct-mapped caches. The arguments in the main body of this article are restricted to single-level caches in uniprocessors. A single-level cache services processor references and obtains data for misses directly from main memory. Most past and present computers use single-level caches, as will many future computers. I expect some future computers, however, to use two-level (or more) cache hierarchies, where a level-one cache services processor references and obtains data for misses from a level-two cache, which in turn services level-one-cache misses and obtains data for its misses from memory. Later in the article, I discuss how my arguments regarding single-level caches extend to two-level cache hierarchies.

I restrict my arguments to uniprocessors for two reasons. First, uniprocessors are and will continue to be important, especially for computers less costly than mainframes, such as engineering workstations. Second, a thorough analysis of caches in multiprocessors requires coverage of many degrees of freedom, which would dilute the thrust of this article. These include interconnection topology, whether control is single-instruction-multiple-data or multiple-instruction-multiple-data, synchronization and cache coherency mechanisms, and number of processors (hence granularity of sharing). I will, however, discuss how and when these arguments for caches in uniprocessors apply to caches in multiprocessors.

Performance metrics

To examine cache performance, I use miss ratio and an extended model of effective access time.

Miss ratio. Miss ratio is the most commonly-used cache performance metric.[1] The miss ratio for a cache C is

$$m(C) = \frac{\text{No. of misses with cache } C}{\text{No. of processor references}}$$

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.

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

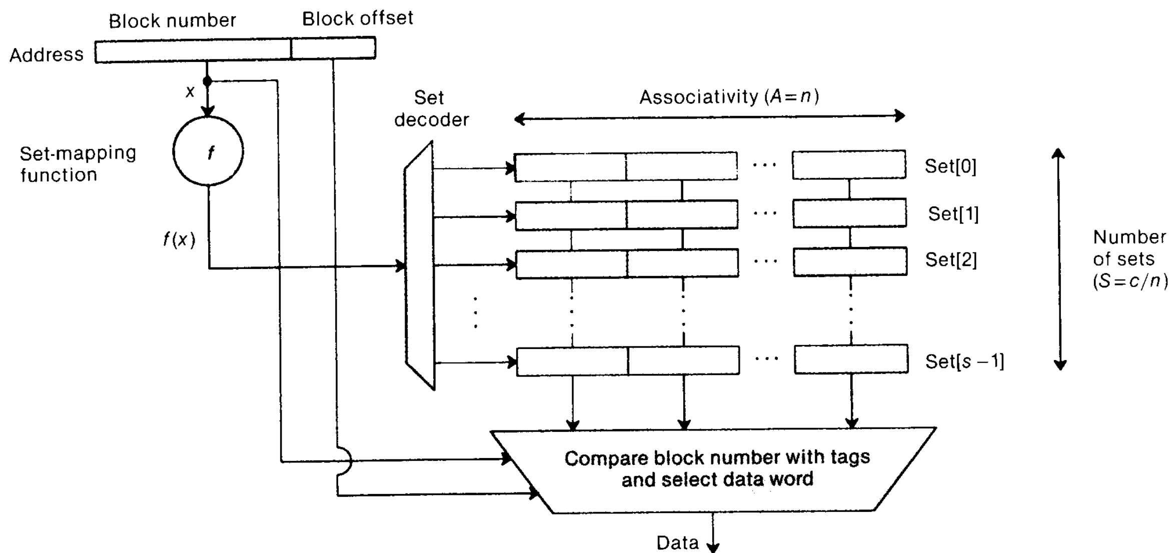


Figure 1. Set-associative mapping. A set-associative cache uses a set-mapping function f to partition all main-memory blocks into equivalence classes. Some cache block frames are assigned to hold recently referenced blocks from each class. Each group of block frames is called a set. The number of groups, called the number of sets (s), equals the number of classes. The number of block frames in each set is called the associativity (degree of associativity, set size, n). The number of block frames in the cache (c) always equals the associativity times the number of sets ($c = n*s$). A cache is fully-associative if it contains only one set ($n = c, s = 1$), is direct-mapped if each set contains one block frame ($n = 1, s = c$), and is n-way set-associative otherwise (where n is the associativity, $s = c/n$).

On a reference to block x, set-mapping function f feeds the set decoder with $f(x)$ to select one set (one row); 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 obtained from memory. Finally, the word requested from block x is returned to the processor. For conceptual simplicity, the figure shows the word selected last (in box labeled "Compare block number with tags and select data word"). To reduce the number of bits that must be read, many implementations select the word while selecting the set.

The most commonly used set-mapping function is the block number modulo the number of sets, where the number of sets is a power of two. This function is called bit selection since it equals several low-order bits of the block number. For 256 sets, for example, $f(x) = x \bmod 256$ or $f(x) = x$ AND 0xff, where mod is remainder and AND is bitwise-and.

block. The actual change in cache performance will depend on how much the number of misses decreases and how much the time to service a miss increases.[3] The penalty for a miss can also vary because of delays indirectly affected by changes in miss ratio, such as memory contention in a multiprocessor.

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

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

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

Using effective access time rather than miss ratio 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 disadvantage, however, is that implementation details must be examined and assumptions must be made for the values of $t_{\text{cache}}(C)$ and $t_{\text{memory}}(C)$. Performance estimates with any implementation assumptions are less general, and those with incorrect assumptions are misleading.

Unlike many other cache memory analyses, my analysis does not assume that $t_{\text{cache}}(C)$ is the same for all caches studied. The disadvantage of including changes in $t_{\text{cache}}(C)$ is that more implementation-dependent parameters must be estimated, further limiting the generality of results. However, variability in cache hit time must be considered, since ignoring it can lead to incorrect conclusions when comparing

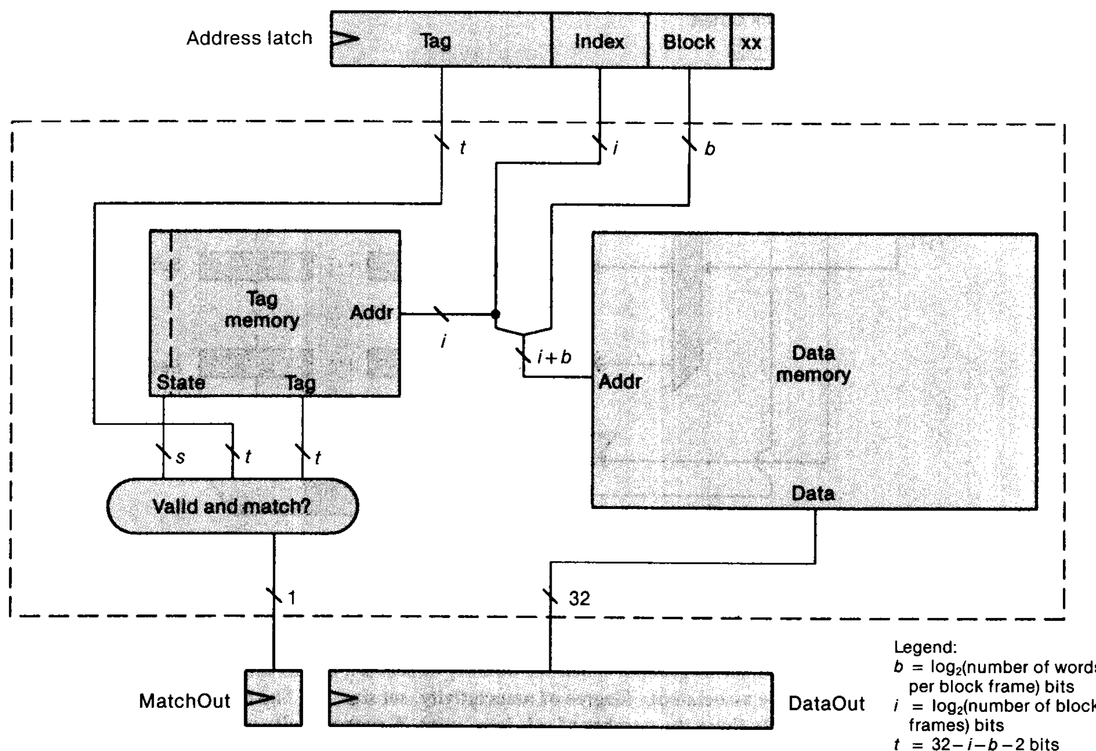


Figure 2. A direct-mapped cache. The access logic (hit, not miss logic) for a direct-mapped cache using bit selection to select the set (block frame) of the reference has three components. The first component, the data memory, holds all cached data and instructions. The second component, tag memory, holds the state bits and address tag associated with a cached block. The last component, the match logic, produces a single bit indicating whether the referenced block is present.

large caches of varying associativities. On the other hand, like many other cache memory analyses, mine assumes that cache changes do not affect the average miss penalty, $t_{memory}(C)$. This assumption simplifies analysis, but it can bias results for multiprocessors where delays due to contention or updating memory are large and variable.

This article does not evaluate caches with system performance metrics like benchmark execution time or effective number of processors, since these metrics require many system-dependent assumptions that 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.

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 four-gigabyte address space of aligned four-byte words addressed with 32-bit byte addresses. I also assume address translation is done in a way that does not affect the cache hit time.

Direct-mapped cache. A direct-mapped cache is simpler to build than a set-associative cache because the cache location of a referenced word is a function of the address of a reference only and the replacement algorithm is trivial. The address of a reference to a direct-mapped cache using bit selection is divided into several fields. From least-significant to most-significant, they are (1) two bits that are ignored, assuming a byte address and aligned word references; (2) $b = \log_2$ (*number of words per block frame*) bits of the block (offset); (3) $i = \log_2$ (*number of block frames*) bits of the index; and (4) $t = 32 - i - b - 2$ bits of the address tag.

Direct-mapped access logic, illustrated in Figure 2, has three components: data memory, tag memory, and match logic. Data memory holds all the cached data and instructions. Its size is, by definition, the cache size. Conceptually, it can be organized as if it were one word wide and accessed with an address formed by concatenating the index (i bits) and block (b

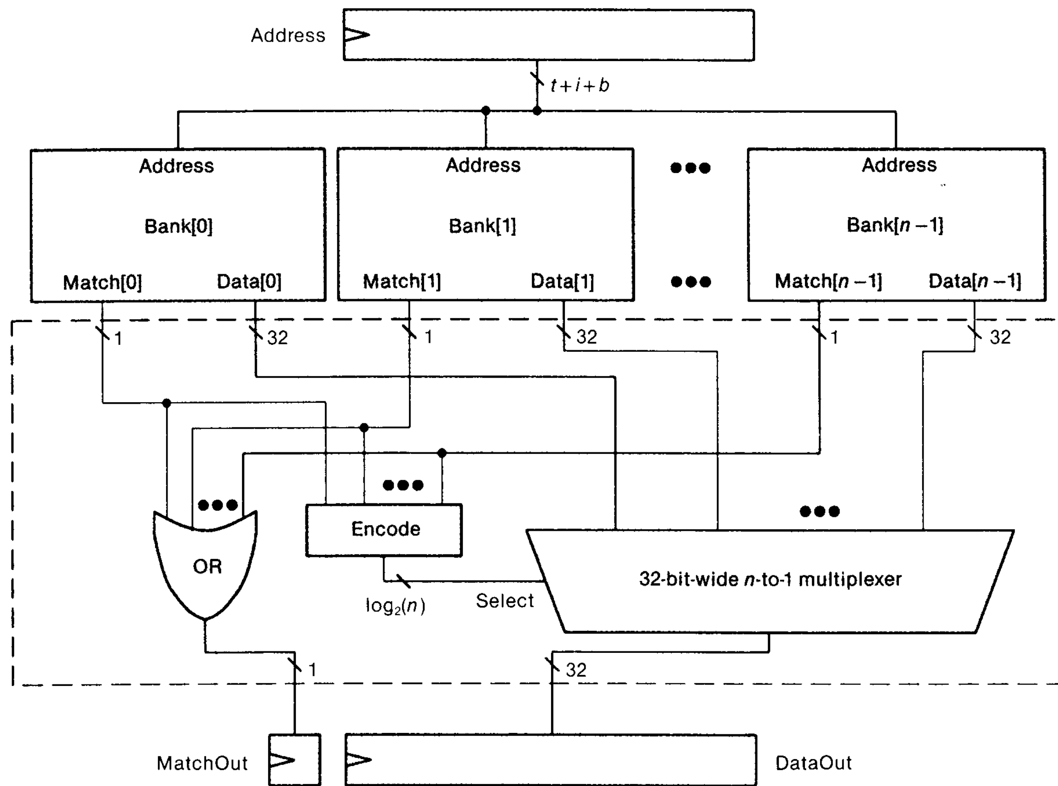


Figure 3. A set-associative cache. Cache access (hit) logic for an *n*-way set-associative cache of *c* blocks consists of *n* banks and the logic to combine bank results. Each bank can be thought of as a direct-mapped cache of *c/n* blocks and can be implemented using the logic in the dashed box of Figure 2.

bits) fields of the address. If it is implemented as a wider memory, some or all of the bits in the block field will be used to select a word after the data memory 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, has one entry per block frame and is addressed by the index field. The state bits for a block, usually one or two bits, indicate the block's status regarding memory update or in a 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.

A direct-mapped cache lookup requires two parallel actions. One action, called read-data, consists of accessing the data memory and passing the word read to DataOut. The second action, called match-found, requires two steps: first, accessing the tag memory to read the state and address tag for a block frame; second, asserting MatchOut if the state is valid and the tag matches the reference's tag.

Thus, a direct-mapped cache lookup is simpler than a set-associative lookup (described below) because actions read-data and match-found can proceed independently. In set-associative caches, the results of match-found influence the data selected.

Set-associative cache. 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 a reference's set to be replaced on a miss. While this flexibility usually yields lower miss ratios, it requires checking n blocks 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). Thus, the index field for each bank requires $i = \log_2$ (*number of block frames/n*) bits, making the tag field, t, $\log_2 n$ bits larger than for a direct-mapped cache of the same size. In addition, some

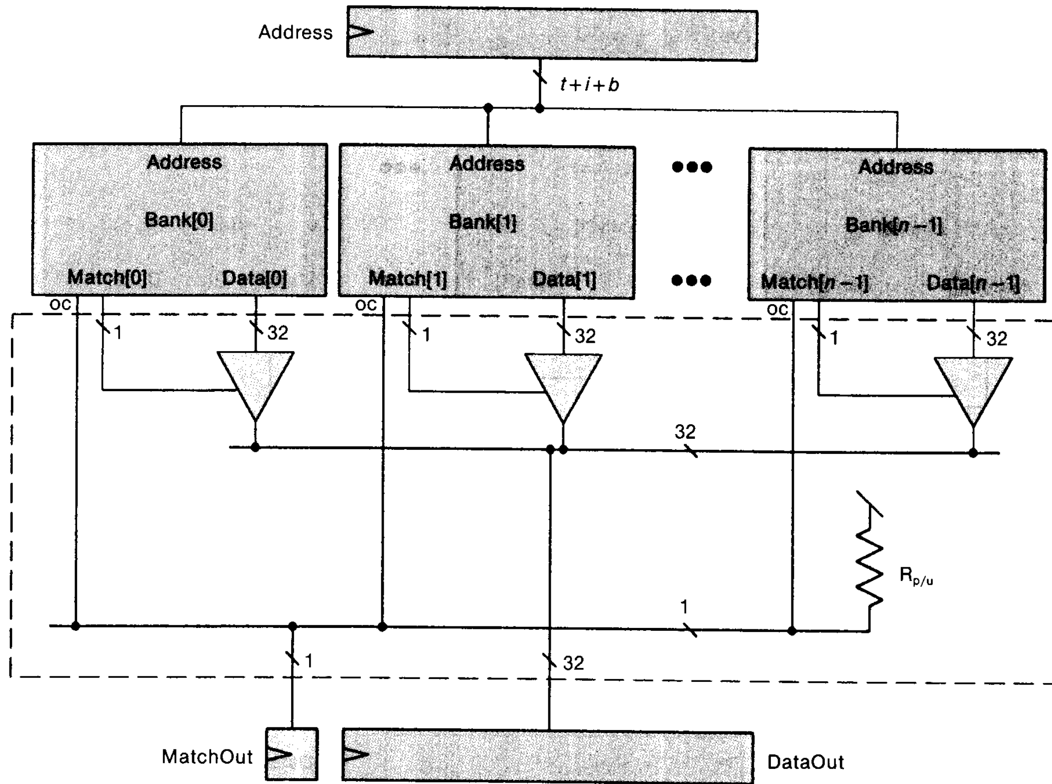


Figure 4. An alternative set-associative cache. This figure shows cache hit logic for an *n*-way set-associative cache with a different set-associativity logic implemention from that of Figure 3. First, it uses wired-OR logic instead of an OR gate to compute MatchOut. Second, the 32-bit-wide *n*-to-1 multiplexer and select logic have been replaced with *n* 32-bit-wide tri-state buffers.

logic, called the set-associativity logic, is needed to select the result from one of the *n* banks.

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 banks compute Match[*i*]'s and Data[*i*]'s, the set-associativity logic, shown in the dashed box in Figure 2, produces a single MatchOut signal and DataOut word. MatchOut, asserted on a cache hit, is the logical OR of the *n* Match[*i*] signals. DataOut, the data to be returned, must be driven to the Data[*i*] for the bank that matched and can be any value if none matched.

One way to implement set-associativity logic is illustrated in Figure 3. Here, MatchOut is computed with a single *n*-input OR gate and DataOut with a 32-bit-wide *n*-to-1 multiplexer. The multiplexer Select input is driven with 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.

Alternate ways of computing MatchOut and DataOut are illustrated in Figure 4. MatchOut is computed by wire-ORing all Match[*i*]'s together, as is possible using open collector (oc) gates in TTL or any ECL gates. This approach requires computing two copies of each Match [*i*] so that the wire-ORing does not affect which data is selected. This duplication does not cause additional delay if the final AND-gate in the bank match logic (not shown) is duplicated.

The alternative implementation for DataOut uses tri-state buffers. Here, each Data[*i*] is connected to the input of a tri-state buffer, whose enable is controled by Match[*i*]. All *n* tri-state buffer outputs are connected together and to DataOut. At most, one tri-state buffer is enabled since, at most, one bank can match. If no banks match, DataOut is undefined.

The distinction between the logic within the *n* banks and the set-associativity logic is not as clear in many implementations as

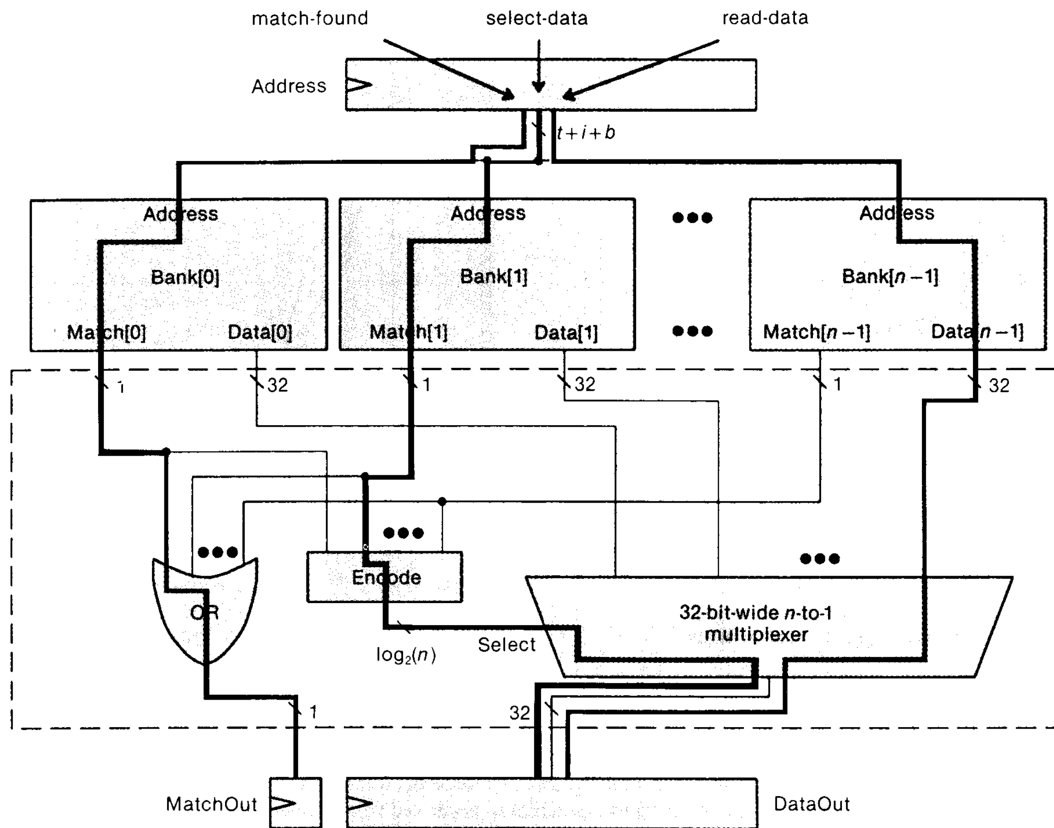


Figure 5. Timing paths in a set-associative cache. The three timing paths in the cache hit logic for an *n*-way set-associative cache are (1) match-found, which signals a cache hit or miss (Address to Match[*i*] to MatchOut); (2) select-data, which selects the data word that corresponds to the tag that matched (Address to Match[*i*] to Select to DataOut); and (3) read-data, which provides the data on a cache hit (Address to Data[*i*] to DataOut). Path select-data is not needed in a direct-mapped cache.

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 multiplexer. Nevertheless, a set-associative cache always requires more circuitry than 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, which signals a cache hit or miss;

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

(3) read-data, which provides data on a cache hit.

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

Arguments against direct-mapped caches

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

Larger miss ratios. It is well-known that direct-mapped caches have larger miss ratios than set-associative caches.[1,4-8] 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

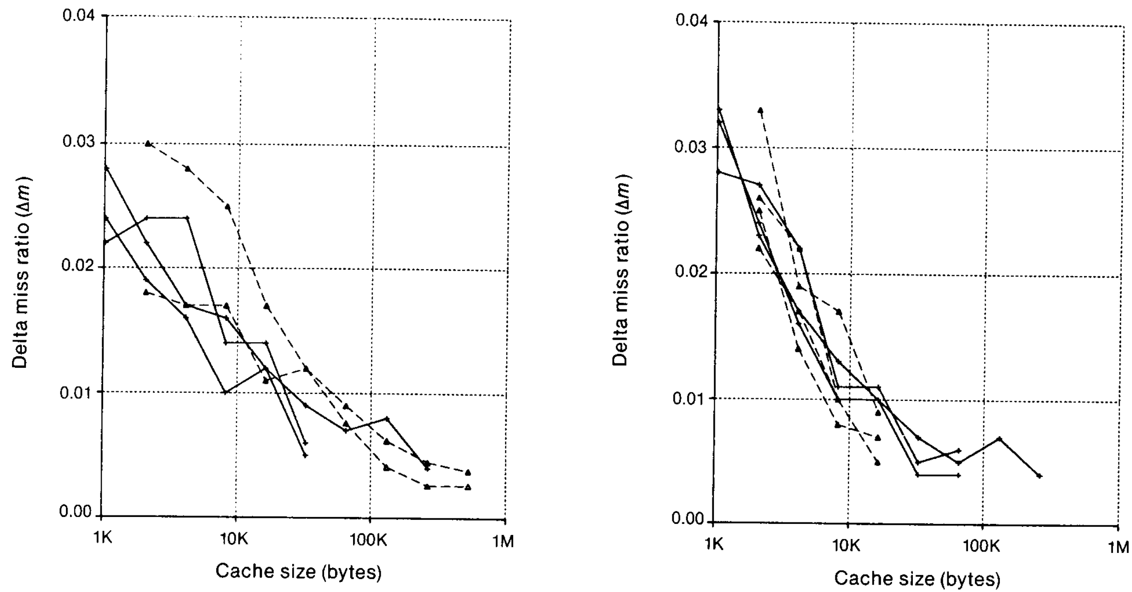


Figure 6. Miss ratio differences for unified caches. This figure shows the changes in miss ratio, Δm, that result when associativity is reduced from two-way to direct-mapped for unified (data and instructions cached together) caches with 16-byte (a) or 32-byte (b) blocks. The data show that miss ratio differences diminish as caches get larger. In comparing 16-byte and 32-byte miss ratios, ignore the dashed lines since this data comes from different traces. The set-associative caches use LRU replacement. (Sources: The miss ratio data in both figures (solid lines) is derived from Tables 2 and 3 in Alexander[6] and Table 3-4 in Hill[9]. Additional data (dashed lines) for 16-byte blocks (a) comes from Figures 5.10a and 5.10b in Agarwal.[7] Additional data (dashed lines) for 32-byte blocks (b) comes from Figures 10-13 in Smith.[1])

ratio difference that results from changing associativity is less than one might expect (Figure 6). The intuition that associativity makes a tremendous difference is wrong, because it fails to consider that references are not made to random locations. Rather, references are usually made to locations in recently referenced blocks. The tendency to re-reference 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 8-Kbyte 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 32 Kbytes the change is 0.005. Miss ratio differences for further associativity increases (from two-way to four-way, from four-way to eight-way), not shown, are much smaller and diminish further as the caches get larger.[9]

Miss ratio differences diminish as 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 show 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 becomes less important for larger caches.

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 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.

Parallel address translation difficult. Almost all high-end computers in the last two decades used paged virtual memory and organized their caches with physical addresses. In these systems, address translation (the translation of virtual addresses to physical addresses) occurs logically before the cache is accessed. For some of these 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, since straightforward implementations require that a cache's size not exceed its associativity times the page size. The IBM 3033, for example, uses parallel address translation and has a 16-way set-associative, 64-Kbyte, physically-tagged cache and 4-Kbyte pages. A 4-Kbyte direct-mapped cache, on the other hand, would not be adequate.

As caches get larger, parallel address translation will become impractical in architectures with fixed page sizes. Eventually the increased hit time and implementation costs of wider associativity will overwhelm the benefits of parallel address translation. Designers will be forced to choose between doing address translation before or after the cache lookup. Address translation is done before the cache lookup on all DEC VAX-11 implementations, for example, since reasonable cache sizes are much larger than the VAX-11's 512-byte page size. Doing address translation after the cache lookup implies that caches are organized with virtual addresses and address translation is necessary only on cache misses. Some researchers argue that the advantage of this approach, namely, a faster hit time, will justify the additional complexity required to implement a cache organized with virtual addresses.[10,11]

In either case, if address translation is not done in parallel with the cache lookup, it will no longer affect whether a cache should be direct-mapped or set-associative.

Arguments for direct-mapped caches

The arguments for direct-mapped caches are (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. Below, I support the above arguments for single-level caches in uniprocessors and show why I expect the direct-mapped organization to become commonly used.

Lower cost. A direct-mapped cache never costs more than a set-associative cache, because there is a way to convert from a set-associative to a direct-mapped design 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 one that is direct-mapped simply by changing the replacement algorithm. On a cache miss, an n-way set-associative cache selects a victim, or block to be replaced, using some algorithm, perhaps 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 one that is set-associative.

In practice, direct-mapped caches cost significantly less, since less parallelism is required if parallel address translation is not done. 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.

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 most 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 (unless the set-associative cache was small enough to allow parallel address translation). The delay paths, displayed in Figure 5, are match-found, select-data, and read-data.

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 address. 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 for 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 at DEC WRL, 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.[12] I found, however, that the performance of a simple direct-mapped cache is similar to that of a more complex MRU cache.[9]

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-state buffers previously used to select data from different blocks.

Finally, improvement in the match-found path is also possible, since it is no longer necessary to read and compare n

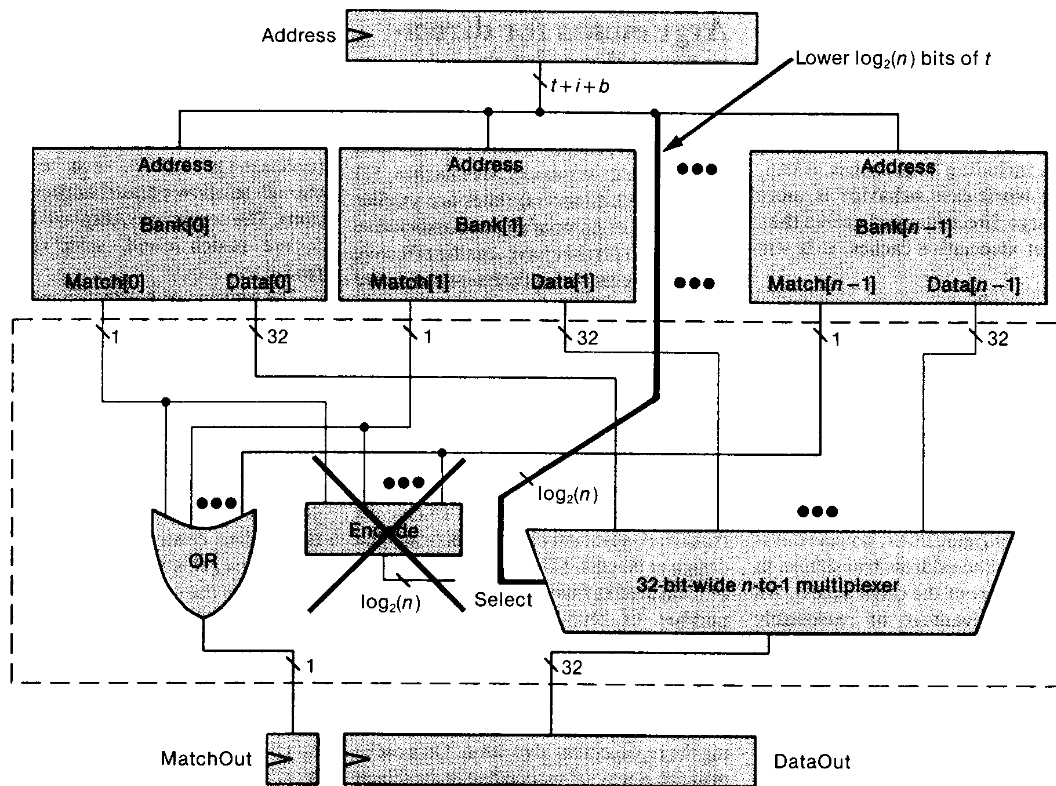


Figure 7. Converting to a direct-mapped cache. An *n*-way set-associative cache can be converted to a direct-mapped cache by changing the replacement algorithm to replace the block in bank *r*, where *r* is the reference's tag modulo *n*. Since this function can be done with bit selection (at trivial 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 increases and, as explained in the text, can decrease cost and hit time.

tags in parallel and 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.

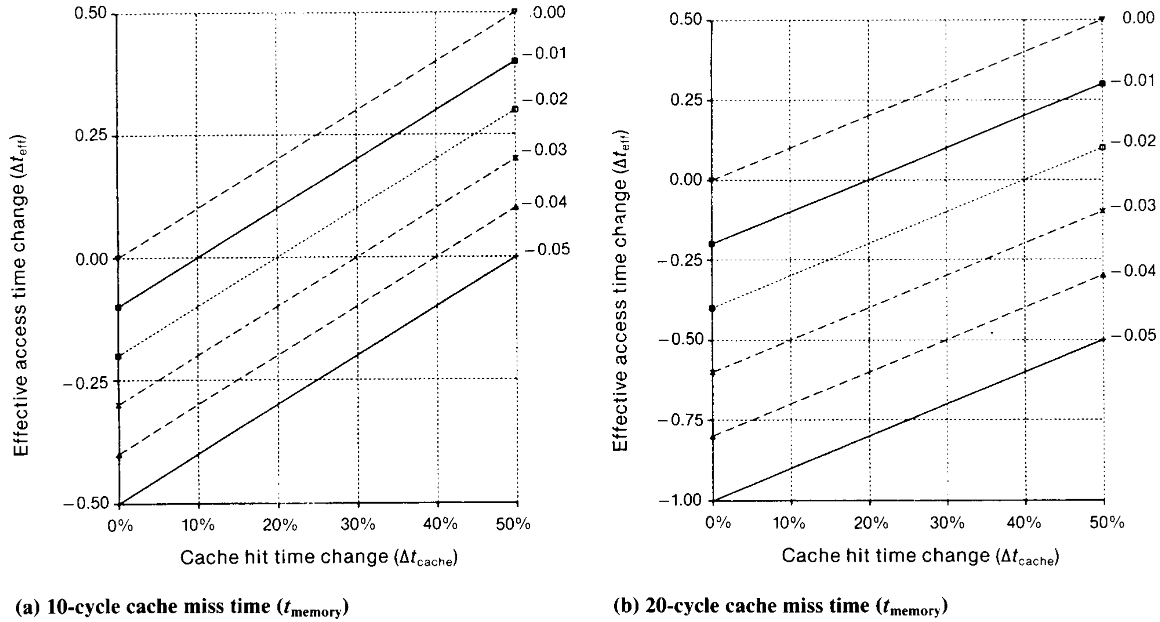
The exact magnitude of the improvement possible depends on many implementation factors. I examined caches implemented in three technologies: (1) TTL logic and MOS SRAM memory chips, (2) ECL logic and memory chips, and (3) custom CMOS. I found that moving from a direct-mapped to a two-way set-associative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

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 between the TTL and ECL times as significant, since both numbers are 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 may 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.

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.

Recall that effective access time, $t_{eff}(C)$, is the average latency, as seen by the processor, required by the memory sys-



(a) 10-cycle cache miss time (t_{memory}) **(b) 20-cycle cache miss time (t_{memory})**

Figure 8. Change in effective access time. This figure shows the change in effective access time ($\Delta t_{eff} = \Delta t_{cache} + \Delta m * t_{memory}$) that results when 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 (a set-associative cache). The graphs assume 10-cycle (a) and 20-cycle (b) 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 percent implies that the slower cache's hit time is 1.2 cycles, 1.2 times the hit time of the 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 cycles. Since most effective access times are slightly larger than 1 cycle, 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 nonpositive, 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 ($\Delta t_{eff} > 0$).

tem to service a memory reference. 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 (cache access time), and average miss penalty (delay beyond a cache access to access memory) 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

$$\begin{aligned}\Delta t_{eff} &= t_{eff}(C_2) - t_{eff}(C_1),\\ \Delta t_{cache} &= t_{cache}(C_2) - t_{cache}(C_1),\\ \Delta m &= m(C_2) - m(C_1), \text{ and}\\ t_{memory} &= t_{memory}(C_1) = t_{memory}(C_2).\end{aligned}$$

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 $\Delta t_{eff} = \Delta t_{cache} + \Delta m * t_{memory}$ 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, typical values must be determined for t_{memory},

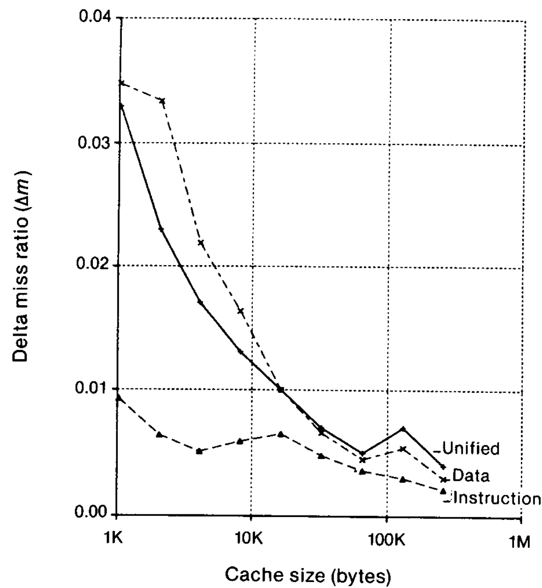


Figure 9. Miss ratio differences. This figure displays the miss ratios from direct-mapped caches less the miss ratio of two-way set-associative caches of the same size for unified, instruction, and data caches with 32-byte blocks using operating system and multiprogramming traces from IBM/370 and VAX 11 architectures.[9] Results show miss ratio differences (Δm) generally diminish with increasing cache size, and are smaller for instruction caches than for unified or data caches.

Δm, and Δt_{cache}. Reasonable values for t_{memory} are 10 or 20 cycles, where a cycle is equal to the hit time of the faster cache. Smaller values are possible, especially in systems where cache misses are serviced by larger level-two 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 16 Kbytes, 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 with cache sizes and positioned according to 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:

(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 64 Kbytes (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 hit time increase is equal to the maximum improvement possible from the smaller miss ratio (for example, 5.2, 3.6, and 4.5 percent for unified, instruction, and data caches of 64 Kbytes, having 10-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 2 Kbytes. The actual benefit will be less if the miss penalty is less than 10 cycles or increasing associativity impacts cache hit time.

Furthermore, increasing block size or increasing associativity beyond two-way does not hurt the case for large direct-mapped caches.[9] Increasing block size in large caches to 64 bytes improves the performance of direct-mapped caches relative to set-associative ones by decreasing all miss ratios and miss ratio differences. Further increases will exhibit similar behavior until the number of blocks in the cache becomes limited. Miss ratio improvements resulting from increasing associativity beyond two-way are much smaller than the improvements between direct-mapped and two-way set-associativity, implying that further increases in associativity will not improve effective access time unless they have a negligible impact on 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 previously, I examined board-level caches (TTL and ECL) where Δt_{cache} was around 10 percent. The effect of a 10 percent slowdown can be studied in Figures 10-12 by only considering design points on a vertical line at Δt_{cache} = 10 percent. For the 10-cycle miss penalty, Δt_{cache} = 10 percent implies that direct-mapped caches have better effective access times than two-way set-associative caches for caches equal to and larger than 16, 8, and 16 Kbytes for unified, instruction, and data caches. For the 20-cycle miss penalty, the corresponding sizes are 64, 16, and 64 Kbytes.

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 cross over is inevitable, given that miss ratio differences diminish as caches get larger and that set-associative caches have

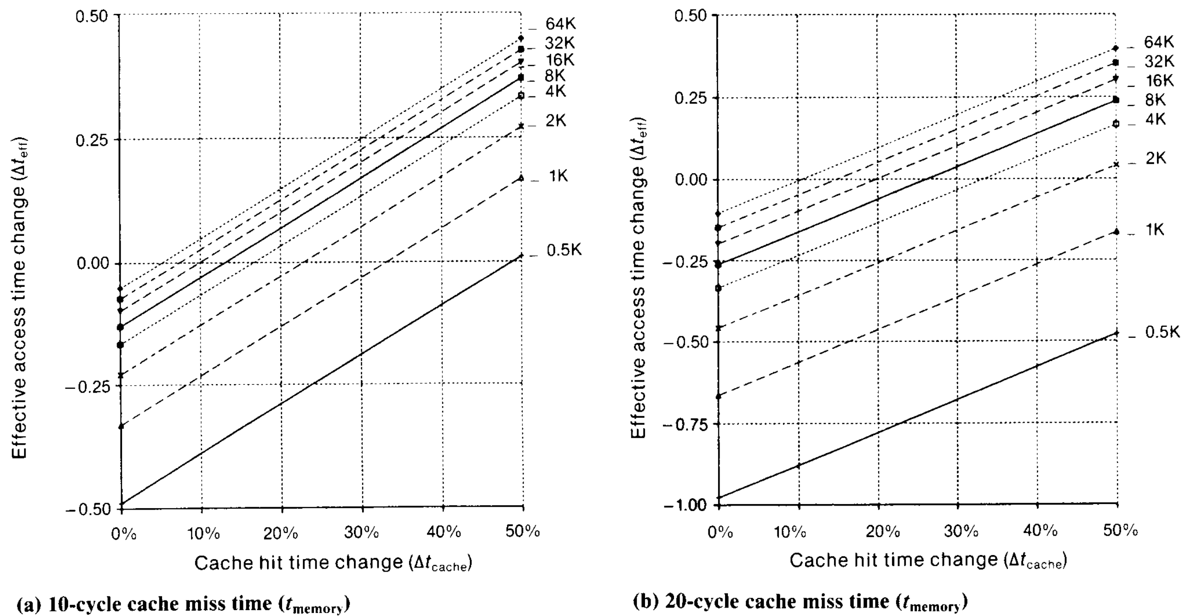


Figure 10. Effective access time changes in unified caches. This figure shows the change in effective access time (Δt_{eff}) that results from moving from a direct-mapped cache to a two-way set-associative cache when both caches are unified, have 32-byte blocks, and have 10-cycle (a) or 20-cycle (b) miss penalties (t_{memory}). This figure is constructed by substituting miss ratio differences (Δm's) 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 16-Kbyte 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 is more than 10 percent slower.

slower hit times.

At cache sizes less than the cross-over size, a direct-mapped cache may still be preferred to one that is set-associative, since a direct-mapped cache may cost less and its effective access time may not be much worse. Even for 20-cycle miss penalties, as Figures 10-12 show, 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 of 32 Kbytes and larger.

Other trends

Up to this point, I have concentrated on single-level caches in uniprocessors. Here I discuss future trends toward caches in hierarchies and multiprocessors. I examine why these trends may occur and discuss how and whether my arguments for single-level caches in uniprocessors apply to these new situations.

Toward caches in hierarchies. In two-level cache hierarchies, a level-one cache services processor references, but it obtains data for misses from a level-two cache instead of memory. A level-two cache services only level-one cache misses and obtains data for its misses from memory.

Two-level cache hierarchies, heretofore rarely used, may become more common in future systems for three reasons. First, implementation considerations can force a partition. Some recently introduced microprocessors, for example, devote some of their limited on-chip area to caches, but they 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 two-level cache hierarchies can offer superior performance to a single-level cache as processors speed up relative to main memories.[8] Third, there may be functional and performance benefits to specializing caches at different levels in a multiprocessor. In a multiprocessor, a level-one cache can be optimized to minimize effective access time, while the level-two cache is designed to reduce cost or

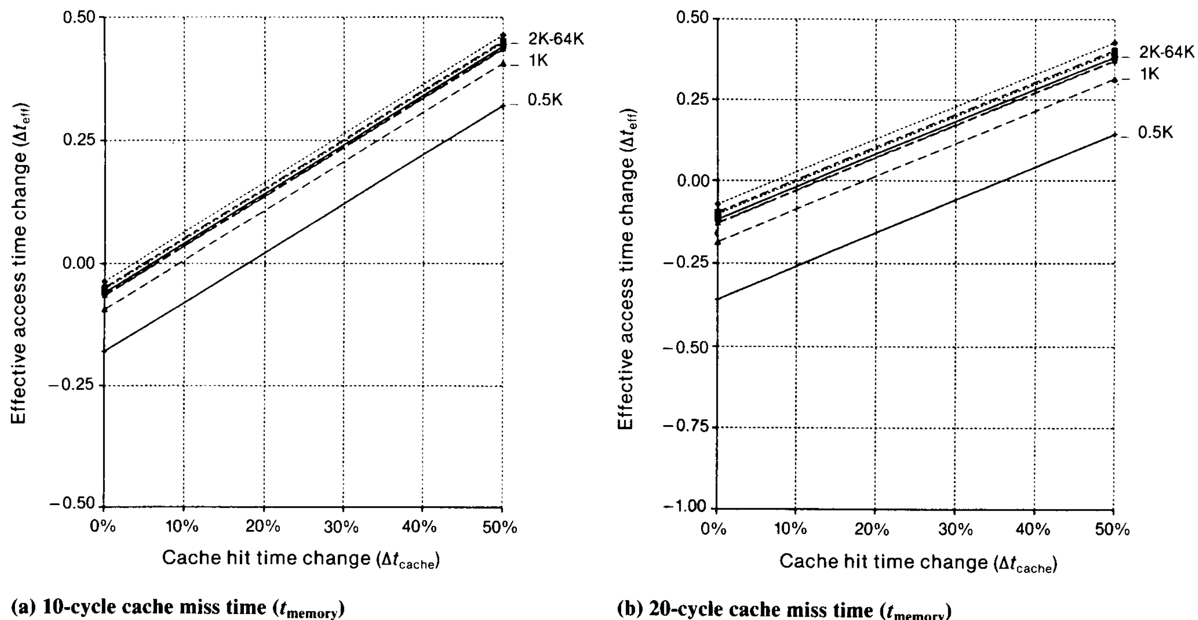


Figure 11. Effective access-time differences in instruction caches. This figure shows the effective access-time change (t_{eff}) of moving from a direct-mapped instruction cache to a two-way set-associative instruction cache with miss penalties of either 10 cycles (a) or 20 cycles (b). Other assumptions match those of Figure 10. Because miss-ratio differences (Δm's) are smaller, the benefit of associativity is smaller for instruction caches than it is for unified or data caches.

interconnection traffic. Similar reasons are expressed by Short and Levy.[13]

The utility of direct-mapped caches in two-level cache hierarchies is, as yet, undetermined. Level-one 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. Direct-mapped caches can be preferred for cache sizes as small as 16 Kbytes if misses are serviced by a level-two cache in 10 cycles or less. Level-two caches, on the other hand, are more likely to be set-associative, since level-two cache hit times are less critical and a lower miss ratio can improve multiprocessor performance. The only argument for direct-mapped level-two caches is that straightforward implementations of large set-associative caches will be expensive, requiring multiple-word-wide banks of memory chips.

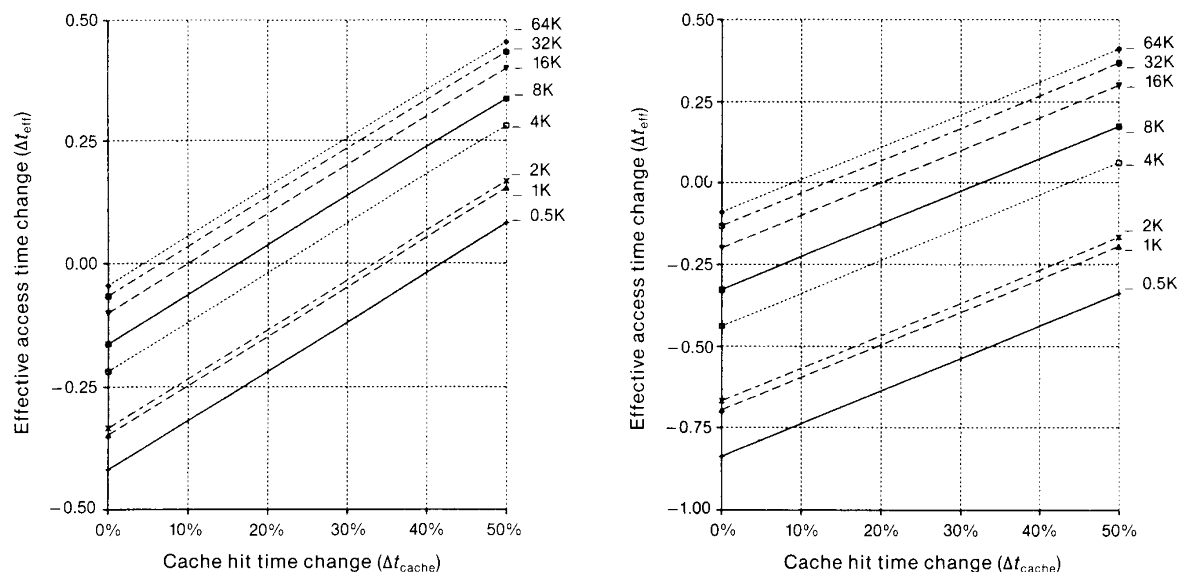
Toward caches in multiprocessors. To provide a rate of growth of computing power that exceeds the rate of technological improvement, many manufacturers, particularly of high-end computers, are turning toward multiprocessors. To facilitate ease of programming, some multiprocessors provide shared-memory and use caches.

Caches in multiprocessors may be designed differently than those in uniprocessors, since multiprocessor caches may be more concerned with minimizing memory and interconnect contention than with minimizing effective access time.[14] Here, the relative miss ratio difference ($\Delta m/m$) is more important than the absolute miss ratio difference (Δm). I found[9] relative miss ratio differences are constant across wide changes in miss ratio and cache size. For example, decreasing associativity from two-way to direct-mapped in unified caches causes a relative miss ratio increase of about 30 percent even for large caches.

Relative miss ratio differences are most important in single-bus shared-memory cache-coherent multiprocessors, where bus bandwidth can easily limit system throughput. In multiprocessors based on long-latency high-bandwidth interconnection networks, however, cache design should proceed as in a uniprocessor with slow main memory. Nevertheless, both cases make set-associativity caches more attractive, but not necessarily better.

If multiprocessors use two-level cache hierarchies, the above arguments apply to level-two caches. I would expect most level-one caches, on the other hand, to be designed like level-one caches in a uniprocessor, making direct-mapped caches likely. This expectation may be incorrect if the misses for many level-one caches are serviced by a single level-two cache and contention between level-one caches is significant.

Direct-mapped caches will be common in uniprocessors as single-level or level-one caches and in



(a) 10-cycle cache miss time (t_{memory}) **(b) 20-cycle cache miss time (t_{memory})**

Figure 12. Effective access-time differences in data caches. This figure displays 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 cycles (a) or 20 cycles (b). Other assumptions match those of Figure 10. The benefit of associativity in data caches is similar to that for unified caches.

multiprocessors as level-one caches. Direct-mapped caches are preferred when they are sufficiently large that hit time benefits are more significant than miss ratio drawbacks. This can occur in single-level caches of 64 Kbytes and larger (16 Kbytes for instruction caches) and can occur at 16 Kbytes and larger for level-one caches whose misses are serviced more rapidly by level-two caches.

The arguments against direct-mapped caches, with respect to set-associative caches, are that they (1) have worse miss ratios, (2) have more common worst-case behavior, and (3) preclude parallel address translation. I have shown that the significance of the first two points becomes questionable for large caches where absolute miss ratio differences are small, and that the third 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 the third point is more likely to be true for large cache sizes.

An alternate way of stating this result is

- set-associative caches reduce the time spent on cache misses;
- direct-mapped caches reduce the time spent on cache hits, especially if a CPU can use data before a hit or miss is determined;
- set-associative caches are preferred in small caches where misses are common;
- direct-mapped caches are preferred in large caches where misses are rare; and
- many future caches will be sufficiently large and therefore direct-mapped.

These arguments may not apply to single-level or level-two caches in multiprocessors, where minimizing contention or very long miss penalties may favor set-associative caches over direct-mapped caches. □

Acknowledgments

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 article: Sue Dentinger, James Goodman, David Patterson, Gurindar Sohi, and the anonymous referees.

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, the State of California under the MICRO program, the graduate school at the University of Wisconsin-Madison, and by IBM, Digital Equipment Corporation, Hewlett-Packard, and Signetics.

References

1. A.J. Smith, "Cache Memories," *Computing Surveys*, Vol. 14, No. 3, Sept. 1982, pp. 473-530.
2. A.J. Smith, "Bibliography and Readings on CPU Cache Memories and Related Topics," *Computer Architecture News*, Jan. 1986, pp. 22-42.
3. A.J. Smith, "Line (Block) Size Choice for CPU Caches," *IEEE Trans. Computers* Vol. C-36, No. 9, Sept. 1987, pp. 1063-1075.
4. J. Bell, D. Casasent, and C.G. Bell, "An Investigation of Alternative Cache Organizations," *IEEE Trans. Computers*, Vol. C-23, No. 4, Apr. 1974, pp. 346-351.
5. A.J. Smith, "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory," *IEEE Trans. Software Engineering*, Vol. SE-4, No. 2, Mar. 1978, pp. 121-130.
6. C. Alexander et al., "Cache Memory Performance in a Unix Environment," *Computer Architecture News*, Vol. 14, No. 3, June 1986, pp. 14-70.
7. A. Agarwal, *Analysis of Cache Performance for Operating Systems and Multiprogramming*, PhD dissertation, Tech. Report CSL-TR-87-332, Stanford University, Stanford, Calif., May 1987.
8. S. Przybylski, M. Horowitz, and J. Hennessy, "Performance Tradeoffs in Cache Design," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, No. 861, Computer Society Press, Los Alamitos, Calif., 1988, pp. 290-298.
9. M.D. Hill, *Aspects of Cache Memory and Instruction Buffer Performance*, PhD dissertation, Tech. Report 87/381, Computer Science Dept., Univ. of California, Berkeley, Calif., Nov. 1987.
10. J.R Goodman, "Coherency for Multiprocessor Virtual Address Caches," *Proc. Symp. Architectural Support for Programming Languages and Operating Systems*, No. M805 (microfiche), Computer Society Press, Los Alamitos, Calif., 1987, pp.72-81.
11. D.A. Wood et al., "An In-Cache Address Translation Mechanism," *Proc. 13th Ann. Int'l Symp. Computer Architecture*, No. 719, Computer Society Press, Los Alamitos, Calif., 1986, pp. 358-365.
12. J.H. Chang, H. Chao, and K. So, "Cache Design of a Sub-Micron CMOS System/370," *Proc. 14th Ann. Int'l Symp. Computer Architecture*, No. 776, Computer Society Press, Los Alamitos, Calif., 1987, pp. 208-213.
13. R.T. Short and H.M. Levy, "A Simulation Study of Two-Level Caches," *Proc. 15th Ann. Int'l Symp. Computer Architecture*, No. 861, Computer Society Press, Los Alamitos, Calif., 1988, pp. 81-88.
14. J.R Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic," *Proc. 10th Ann. Int'l Symp. Computer Architecture*, No. M473 (microfiche), Computer Society Press, Los Alamitos, Calif., 1983, pp. 124-131.

University of Nebraska - Lincoln
Computer Science and Engineering Department
Department Chair

The University of Nebraska-Lincoln seeks a dynamic individual for the position of Chair of the Department of Computer Science and Engineering. The department currently has 14 faculty members, has in place rigorous programs in computer science and has initiated a program in computer engineering. Programs are offered in two colleges, Arts and Sciences and Engineering and Technology. Currently about 380 Undergraduates, 55 Masters and 15 Ph.D. candidates are enrolled in Computer Science and Engineering programs.

The department has strong research programs in algorithms, theoretical computer science, communications theory and networks, coding theory and data encryption, combinatorics, fault tolerant computing, formal languages, and symbolic and algebraic computation. Research strengths also exist in artificial intelligence, computer architecture, VLSI, programming languages, numerical analysis, information retrieval, human factors, and data base.

Strong interdisciplinary ties exist between the Departments of Computer Science and Engineering; Mathematics and Statistics; Electrical Engineering; and Computer Resources Center. The University has recently created a *Center for Communication and Information Science* based mainly on research faculty in the Computer Science and Engineering Department, but also including faculty from the above named departments. The new chairperson will have a leadership role in shaping the Center's future direction.

The University of Nebraska - Lincoln is the primary campus for research and graduate studies in the State of Nebraska. The University has a wide variety of computing resources linked by a sophisticated campus-wide network. UNL is the leading institution in the NSF-funded regional network MIDnet, and a node on the NSFnet backbone.

The State of Nebraska's commitment to technology has been underscored by the Governor's Proposal to increase funding for research at the University of Nebraska. The five year plan would provide an additional $4 million each year over the previous year, leading to a $20 million increment in the fifth year. The State Legislature has appropriated funds to start this ambitious project. Some of these funds are now available to support the Center for Communication and Information Science.

Qualifications require earned doctorate in computer science or related field, strong leadership for research and academic programs, and credentials appropriate for appointment as a full professor. Administrative experience is desirable.

The starting date for this appointment is August, 1989. The closing date is December 15, 1988, or until the position is filled. Salary will be commensurate with qualifications. Women and minorities are particularly encouraged to apply.

Qualified applicants should send resumes and names of three references to Prof. Spyros S. Magliveras, Chairman, Search Committee, Computer Science and Engineering Department, Ferguson Hall, University of Nebraska, Lincoln, NE 68588-0115. E-mail address: spyros@fergvax.unl.edu.

An Equal Opportunity/Affirmative Action Employer

Mark D. Hill is an assistant professor in the Computer Sciences Department at the University of Wisconsin at Madison. His research interests center on performance and implementation factors in memory systems. He was a principal contributor to the SPUR project to build a shared-bus multiprocessor at the University of California at Berkeley. He is currently working on Multicube, a project designing a multiprocessor using a grid of buses.

Hill earned a BS in computer engineering from the University of Michigan in 1981, and an MS and PhD in computer science from the University of California at Berkeley in 1983 and 1987, respectively. He is a member of IEEE, the IEEE Computer Society, and ACM.

Hill may be contacted at the University of Wisconsin-Madison, Computer Sciences Department, 1210 W. Dayton St., Madison, WI 53706.