

Cache Memory Design: An Evolving Art

Alan Jay Smith

University of California, Berkeley

Designers are looking to line size, degree of associativity, and virtual addresses as important parameters in speeding up the operation

Cache memories—buffers that hold information from main memory so that a processor can reach it quickly—have long been an essential part of mainframes and of many minicomputers. Now they are being built into systems using microcomputers. They are being incorporated in one-chip microcomputers as on-chip caches that make access time even shorter. Some processor-chip manufacturers are even building their products around cache memories, making the system designer's job simpler.

Although cache memories have been available since first introduced in the IBM 360/85 computer in 1969, their design is far from cut and dried. The designer has to make several choices and set various parameters. For example, decisions must be made on the algorithms that fetch, place, and replace information, on the way the cache should be addressed, on the best size of the cache, given performance goals and design constraints, and on the best way to ensure consistency among several caches in a multiprocessor. Designing a cache properly is tricky, but getting it right is crucial. Performance of most computers strongly depends on the quality of the cache design and the way in which it is implemented.

Caches continually obtain and temporarily retain items the processor is likely to need in its current operations. They function far faster than does main memory, which is designed to store large amounts of information economically, rather than for speed of access. For example, a large, high-speed computer such as the Amdahl 580 or the IBM 3090 takes 200 to 500 nanoseconds to access main memory, but only 20 to 50 ns to access cache memory.

Designing the best cache for the job

Caches work on the principle that a processor is likely in the near future to need the information it is working on at the present, along with information lying nearby in its main memory [Fig. 1]. A cache attempts to ensure that such information, local both in time and in space to the current information, is readily available.

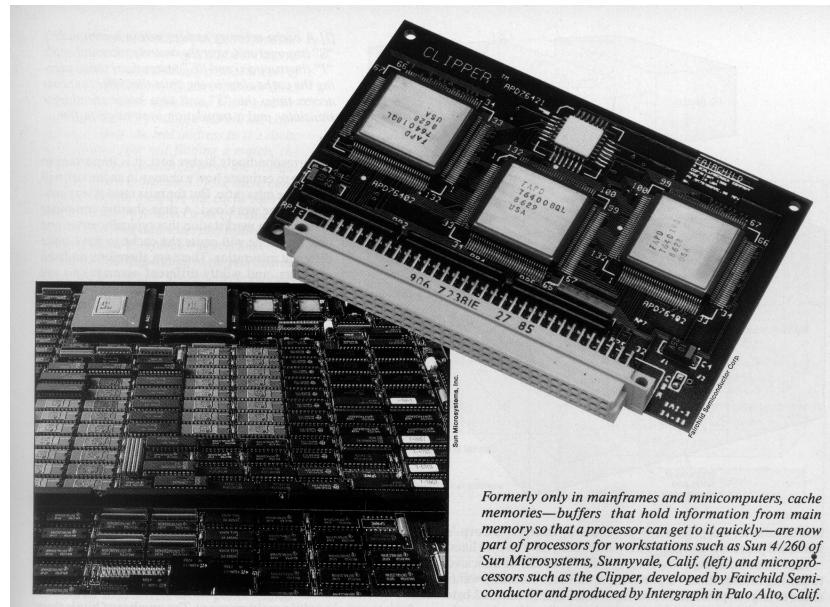
The success of a cache design can be evaluated quantitatively by two primary factors:

- The probability of finding the needed information in the cache (making a hit) instead of having to go to main memory (a miss): that probability is given as the hit ratio, which is equal to 1 minus the fraction of cache references that miss (the miss ratio).
- The mean time it takes to access the cache in a hit.

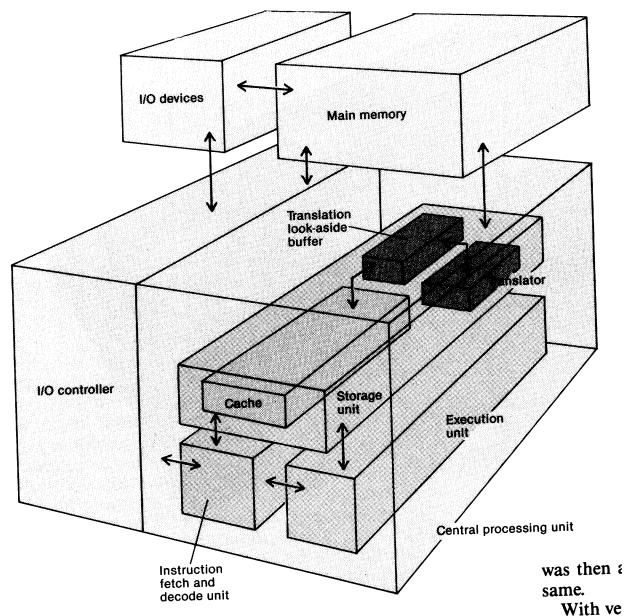
Besides aiming for a hit ratio as close to 1 as possible and a mean access time as low as possible, designers want to optimize several important secondary factors:

- Reduction of main-memory access time in case of a miss.
- Reduction of the total information demanded in a multiprocessor system so as to reduce queues at main memory.
- Elimination of any cache cycles lost in maintaining consistency among multiprocessor caches.

Usually small changes in the secondary factors produce a much smaller effect on system performance than small changes in the hit-ratio and the cache-access time.



Formerly only in mainframes and minicomputers, cache memories—buffers that hold information from main memory so that a processor can get to it quickly—are now part of processors for workstations such as Sun 4/260 of Sun Microsystems, Sunnyvale, Calif. (left) and microprocessors such as the Clipper, developed by Fairchild Semiconductor and produced by Intergraph in Palo Alto, Calif.



[1] A cache memory resides within a computer's "S" (storage) unit, near the central processing unit's "I" (instruction) and "E" (execution) units, placing the cache close to the units that refer to it cuts access time, the "S" unit also holds an address translator and a translation look-aside buffer.

Defining terms

Arbitration time: the time taken to determine which of simultaneous contenders for a service takes priority.

Associativity: the number of information elements per set in a cache.

Constituency (coherency): agreement between the shared contents of caches.

Line (block): the basic unit of information exchange, consisting of one or more information elements, between a cache and main memory.

Line crossers: references to memory that span the boundary between two cache lines.

Page crossers: references to memory that span the boundary between two pages.

Page fault: a trap that occurs when data requested by a processor is not found in main memory.

Page table: a data structure that translates virtual addresses into real addresses.

Real address: the actual hardware address of an information element in memory.

Transfer time: the time required to move an information element from memory to a processor.

Virtual address: an address generated by a program and later translated into a real address for the main memory.

A TLB may be set-associative or fully associative. A set-associative TLB partitions the entries into two or more sets of entries, using the lower bits of the page number to select a set, and searches that set for the required translation. A fully associative TLB searches its entire contents. In either case, when the TLB finds a match for the virtual address, it passes the corresponding real address to the cache.

The TLB has limited capacity, however—typically anywhere from 4 to 512 entries—and may not hold the match for a given virtual address. When that happens, it refers to the translator, a part of the machine that uses page tables in main memory to translate a virtual address into its real address. The translator sends the real address to the TLB, which retains a copy of it for a possible reuse, forwarding it to the cache. When the TLB is full, it discards an old address translation to make room for the new one. The translator is much slower than the TLB, and using it adds time to the information retrieval process.

The byte-within-page portion of the address has two parts: the line number, and the byte within the line [Fig. 3]. While the TLB starts the translation process, the cache is using the line number—the middle part of the virtual address—to select a set (grouping) of entries in a set-associative cache. Each entry in the cache consists of a real-address tag that identifies the data, and the line of information stored at that address. The cache compares the addresses of the entries with the real address from the TLB and selects the line of information whose address tag matches the real address.

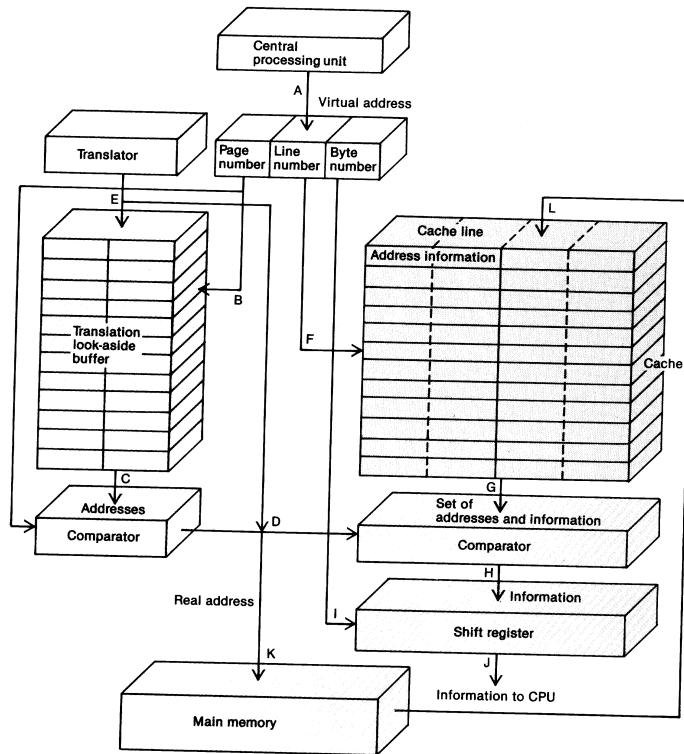
When the proper line has been found the cache selects the bytes it actually wants, using the byte-within-line part of the address. The cache forwards the chosen bytes to the CPU, and the retrieval process is complete. The cache updates its record of recent uses to note that the address has been used.

The central processing unit can also request that information be written into the cache rather than read from it. The virtual address will then be accompanied by a control signal indicating a write, but the sequence of events is similar. If the cache does not contain the address wanted—if it cannot find an entry for the real address—it handles a miss by sending the real address to main memory, which responds by returning the corresponding information line.

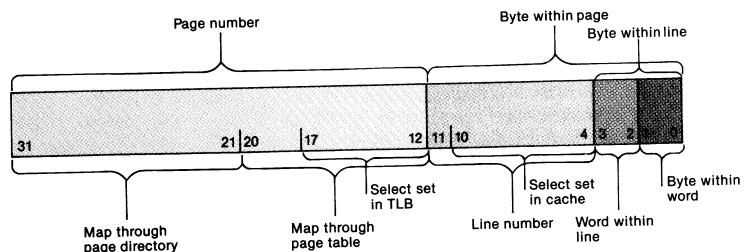
While waiting for main memory's response, the cache makes room for the new entry, reviewing its usage records and deleting an information line that has seen little recent use. Before it makes the deletion, however, the cache checks to see whether the CPU has modified the line since it was withdrawn from main memory. If such modification has taken place, the cache replaces the original line in main memory with the modified line.

How a cache works

When a central processing unit needs information from its cache memory, it sends the virtual address of the information to the cache and to a translation look-aside buffer (TLB), a small, fast, associative memory that stores recently used pairs of virtual and real addresses [Fig. 2]. Addresses, real or virtual, consist of a page number and a byte position within the page. The page number needs to be translated from virtual to real; the byte within the page is the same for both virtual and real addresses.



[2] To access information, a central processing unit generates a virtual address (A) the translation look-aside buffer selects a set of address translations (B); a comparator seeks a match between translation entry and virtual address (C); finding a match, the comparator sends the real address to the cache comparator (D); not finding a match, the TLB gets the real address from the translator(E), which is then used by the cache comparator to find a match, the cache uses the line number (F) to select a set of associated lines containing addresses and information (G) the cache comparator selects lines of information corresponding to the real address (H) a shift register uses the byte number from the virtual address (I) to select the required information and sends it to the CPU (J); the cycle is then complete. If the cache does not contain the desired information, it sends the real address to main memory (K); main memory sends the required information back to the cache (L), which forwards it to the CPU.



[3] The major components of a virtual address are the page number and the address of the byte within the page. Both components are further broken down into subcomponents. For example, the lowest bits of the page number (here bits 12 to 17) select a set of entries in a translation look-aside buffer for comparison with the higher bits in the page number (here 18 to 31) to find a match between the virtual address and a real one. Other subcomponents, such as line number, word-within-line, and byte-within-word, zero in on the required information. The subdivisions shown are those for the 32-bit virtual address in the cache in the Clipper system.

Specifying parameters

With such a complex operation, it is not surprising that a designer has to make many tradeoffs in setting line size, cache size, the degree of associativity, and so on. Further questions include whether to use real or virtual addresses to reference the cache, when to update main memory with modified data, whether to split the cache into parts, how big to make the cache, whether to make the cache hierarchical, and how to maintain consistency if there is more than one cache.

The length of a line affects the amount of delay from cache misses. A line (also called a block) is the basic unit of information transfer between cache and main memory, and its size also affects the miss ratio. As the line grows from very small to very large, the miss ratio will at first decrease, since a miss will fetch more data at a time. The miss ratio will increase, however, as the line becomes even bigger and the probability of using the newly fetched information becomes less than the probability of reusing the information that has to be replaced.

To determine the effect of line size on system performance, the designer considers the time to service a miss: the line size multiplied by the transfer time per byte, added to the overhead time for each miss. Usually that overhead time—which ranges from 100 to 400 ns against transfer times of 10 to 25 ns per byte for common-bus systems—is much greater than the byte transfer time. To keep the time to service a miss as low as possible, the designer chooses the line size yielding the lowest average delay per memory reference. The optimal size depends on the cache size and access time parameters and ranges from only 4 bytes for a small cache (32 bytes) to as many as 128 bytes for a large cache (128 kbytes). The two 4-kbyte Clipper caches developed by Fairchild Semiconductor Corp. (and now produced by Intergraph Corp.'s advanced processor division, Palo Alto, Calif.) have 16-byte lines.

Line size also affects how much of the cache can be allocated to storing information—a portion the designer naturally wants to see as large as possible. For example, if a 6-byte line has a 2-byte address, the address occupies one-third of the line, therefore only two-thirds of the cache contains useful information. But if the line is 64 bytes long, the 2-byte address takes up only 1/33 of it: the remaining 32/33 of the line is available for information.

Another factor to consider is the frequency of line crossers (memory references that span the boundary between two cache lines) and page crossers (references that span the boundary between two pages). Because they are usually handled as two accesses, line crossers take additional time and, of course, the shorter the

lines, the more line crossers there will be. Page crossers are a special problem in the design of cache and of CPUs, because of the possibility of page faults on either half or even in both halves of the line. Such a possibility of page faults can significantly complicate the CPU pipeline design, since each page fault is a trap and must be handled correctly.

Yet another factor is that longer lines mean longer tie-ups at the memory interface, slowing down a multiprocessor system as one processor locks out the others from shared memory while it handles a miss. Correspondingly, a miss can cause I/O overruns, when I/O devices exceed their own buffers while waiting to reach main memory, leading to further delay.

How big should a cache be?

Cache size, like line size, strongly affects the miss ratio, but the size of the cache is subject to constraints. Clearly, the larger the cache, the lower the miss ratio and, other things being equal, the better the performance. But the larger fan-in and fan-out of the gates in a high-capacity cache introduce longer rise times, and the large caches thus tend to be slightly slower than the small—even when built with the same integrated-circuit technology and put in the same place on chip and circuit board. Cache size is also limited by the available chip and board area, and larger caches cost more and need more power and more cooling.

Because larger caches deliver better performance, even though at correspondingly higher cost, it is important to be able to estimate how a change in cache size will affect the miss ratio. But the miss ratio is very sensitive to the workload. A time-sharing computer system and a workstation that typically serves one user at a time will cause the cache to have quite different miss ratios. There are, therefore, no fixed numbers, and wildly different assumptions are made by various designers. One solution developed by this author is to use design-target miss ratios.

Toward a cache hierarchy

As integrated-circuit technology advances, designers will more and more favor large, multilevel caches. Until recently, a cache was just one level in a multilevel-memory hierarchy, comprising cache, main memory, and auxiliary storage such as disk and tape. Because they had to be implemented as collections of chips, with only the medium-scale or large-scale integration—all that was then available—each chip cost and performed about the same.

With very large-scale integration, however, caches no longer have to be multichip and can be built on a

single chip, or even as part of the microprocessor chip itself, as in the Motorola 68020 and 68030 and the National 32532. It even becomes feasible to add levels to the cache by adding slower, larger, and cheaper chips for the information used less frequently, but still too frequently for it to remain only in main memory. The number of hits is then greatly increased at little extra cost.

In the 68020 cache, the miss ratio is more than 50 percent, which can be brought down by adding off-chip cache capacity for an inexpensive backup to cut the misses. For the same reasons, the Clipper cache's 10 percent miss redo justifies in high-performance multiprocessors a large board-size cache of about 256 kbytes.

Chip technology is yet another factor affecting the size of a cache. Naturally, the designer would prefer the fastest technology available, and while emitter-coupled logic offers access times of less than 6 ns, CMOS currently gives at best 10 to 12 ns. But fast ECL chips eat up power, are costly, and have low information density.

Designers therefore do not often choose the fastest technology. Price drops rapidly with performance, while density goes up. The fastest CMOS static-RAM chips can hold 16 or 64 kbytes, but somewhat slower dynamic RAMs can hold as much as 1 megabit.

The same speed-density-cost tradeoff applies to on-chip caches: CMOS technology could currently fit 32 kbytes of DRAM—or 8 kbytes of SRAM—into 25 percent of the processor chip's area.

Fetching and replacing

Most caches fetch on demand and bring a line from main memory only when it has been called for by the CPU and missed. An alternative is pre-fetching, in which lines are fetched when the cache anticipates they will be needed. Sequential pre-fetching, for example, fetches the next line in sequence to a line already fetched, a routine that on average cuts the miss ratio in half—provided the cache is large enough.

The obvious strategy for making room for new information is for the cache to discard the line that has been least used recently and replace it with the new line. The least-recently-used algorithm works well, as does an approximation of such an algorithm; it is also possible simply to discard a line at random—or even on the first-in, first-out basis—with only small loss in performance.

Organizing the cache

Most caches are set associative. They map an address into a set (usually by using n of the line number bits to select one of the 2^n sets) and search associatively within

the set for the correct line. The degree of associativity (the set size) affects cache performance. If there is only one set, the cache is fully associative, and if there is only one element per set, it is called direct-mapped.

As might be expected, increasing associativity—the number of elements per set—cuts the miss ratio, but only up to a point. Two-way associativity—a set size of two elements—is significantly better than direct mapping, and four-way is better yet, although only slightly; further increases have little extra effect.

But increasing associativity requires more components and connections to read out and compare addresses; or, if the cache is on one chip, the added complexity requires more silicon area. Going from direct mapping (one element per set) to two or more elements requires an extra level of logic, which adds delay. The result is that for large caches (for example, above 32 kbytes), where the miss redo is already low, a direct-mapped cache's shorter access time gives better performance than does a set-associative design. Conversely, for smaller caches, where delays from misses dominate, the set-associative design is preferable.

Keeping main memory up-to-date

When the CPU modifies information in a cache, the new information will sooner or later have to be placed in main memory, either by being immediately transmitted (write-through) or by being sent from the cache when the line containing the information is replaced (copy-back).

Write-through is simpler and ensures that the main memory is kept up-to-date. It is also slightly more reliable, because main memories usually have elaborate error-detection and correction mechanisms, whereas caches can seldom detect errors, let alone correct them. But write-through generates substantial memory traffic and may create a bottleneck; copy-back generally yields better overall performance.

The trend to shared-memory multiprocessors makes write-through bottlenecks a serious problem, and copy-back caches are gaining ground. Nevertheless, it is not particularly difficult to have both write-through and copy-back in the one cache.

Write-through is still the better choice, however, when a microprocessor has a small on-chip cache as well as a larger off-chip cache. To ensure consistency between the caches, information should be written through the on-chip cache into the off-chip cache.

Virtual-address caches

With most caches, the virtual address generated by a CPU must be translated into a real address before the

data can be located in the cache. It does not have to be that way, however: it is possible to access a cache directly with the virtual address. Although very few machines, the Amdahl 580 being one, have a virtual-address cache, such a design presents advantages. For one thing, no translation is needed for cache use, so access is faster. (For references to main memory, virtual-to-real translations are still needed, but they would be done only when needed and do not have to be done quickly.)

Secondly, there is no need to attempt to overlap the translation and cache-access processes to save time. Virtual-address caches are tricky to design, however, because it is possible that two virtual addresses—called synonyms—can be translated to refer to the same real address. Handling such a conflict may lead to complexities in the software and hardware.

Synonyms can occur in several situations:

- When two different programs on the CPU, one of which may be the operating system, share pages placed in different places in the two programs' respective address maps.
- When a program has asked the operating system to assign different virtual addresses in the program's own address space to the same real address.
- When an I/O device, using real addresses, refers to a region of main memory also accessible to a program.
- When a program running on a CPU shares the same memory as a program on another CPU, and has some part of its virtual-address space mapped onto the same memory locations as the other program.

The synonym problem can be solved by adding a reverse translation buffer (RTB), a logic circuit that maps real addresses back to virtual addresses. RTBs are complex and not particularly easy to add to a system. Nevertheless, it seems likely that as caches become larger and demands for higher speed grow more insistent, virtual-address caches will become more popular.

One cache or split?

Traditionally, computers have been built with a single cache for both data and instructions. This is the simplest method—one cache communicates with one main memory. Moreover, the CPU's components have only one unit to refer to, whether they are reading or writing data, or calling for instructions. In addition, sharing the same unit for data and instructions makes more effi-

cient use of a limited resource and so lowers average miss ratios.

But there are advantages to splitting data and instructions into two separate caches. Conflicts between simultaneous instruction fetches and data reads and writes are eliminated, as is, consequently, arbitration time; and each cache can be placed near the CPU component that most often needs it, which further reduces access time. These issues are closely related to CPU pipeline design, and are not solely tied to cache design.

The main impediment to split caches has been the need to ensure that writes into the instruction stream actually have the desired effect. New architectures, however, such as those in the Clipper system, National 32000, and Motorola 68000, can prohibit such writes and can therefore more easily use split caches. In older architectures, such as the IBM 370, such a prohibition is impossible, and building split caches requires a means to write into the instruction cache.

Cache consistency

Programming algorithms on multiprocessors is much easier when the processors share a common memory. When individual processors have individual cache memories, however, the caches may hold different versions of shared data. The system must include something—called a consistency (or coherency) mechanism—to ensure that does not happen. Several approaches are possible, including smart memories and software control. The trend for systems with small or moderate numbers of processors is toward bus-based consistency; systems with very large numbers of processors tend to software-consistency control.

With a common bus, all processors share the same path to main memory. Each cache monitors misses and writes-to-memory by all other caches, and each maintains a directory of its own information and records whether any other cache holds the same data. When a processor does something to data that affects—or could affect—a copy of the data held by another cache, the local cache must broadcast on the bus all information necessary to maintain consistency. This might mean that the other cache gives up its copy of the data, updates its local copy, or provides the correct copy on a read.

Ensuring consistency with a common bus has its limitations and difficulties, chief among them greater complexity, and wide-spread use will require special VLSI chips. The principal limitation is that even the best bus-based cache-consistency mechanisms require a common bus for all memory traffic. And even though a good consistency protocol reduces memory traffic to well below that required for write-through, the amount

of traffic will still limit the number of processors that can share the bus.

With software control of consistency, the operating system must know which areas of memory are shared and when. The operating system issues commands to the various processor caches to ensure the shared information is kept consistent, and it will make a processor purge its cache if another processor has modified shared memory. The problem here is that synchronization is critical and frequent purging slows the system.

Cache consistency is such an active field of research that computer designers are likely to come up with simpler and more effective ways of ensuring consistency over the next few years.

To probe further

Cache memory is covered in the text books *High-Performance Computer Architecture*, by Harold Stone (Addison-Wesley, 1987), and *High-Speed Memory Systems*, by A.V. Pohm and O.P. Agrawal (Reston, 1983). For a survey of the technology, see "Cache Memories," in *ACM Computing Surveys* (Association for Computing Machinery), September 1982, pp. 473-530. For a discussion of cache workloads and how they affect cache design, see "Cache Evaluation and the Impact of Workload Choice," in *Proceedings of the 12th International Symposium on Computer Architecture*, Boston, June 17-19, 1985, pp. 64-73 (IEEE Publications No. 85CH2144-4).

Cache memory design is covered in "The Memory System of a High-Performance Personal Computer," by D.W. Clark, B.W. Lampson, and K.A. Pier, *IEEE Transactions on Computers*, Vol. C-30, no. 10, October 1981, pp. 715-733.

For details on the performance of caches as a function of line size see the present author's "Line (Block) Size Selection in CPU Cache Memories," in *IEEE Transactions on Computers*, Vol. C-36, no. 9, September 1987, pp. 1063-1075; the same paper provides design-target miss ratios for a variety of cache and line sizes.

Cache performance in a time-sharing system is presented in D. Clark's, "Cache Performance in the Vax 11/780," *ACM Transactions on Computer Systems*, February 1983, pp. 24-37.

A key early commercial cache implementation is described by J. Liptay in "Structural Aspects of the System/360 Model 85: Part II: The Cache," *IBM Systems Journal* 1968, pp. 15-21.

About the author

Alan Jay Smith (SM) is a professor of computer science at the University of California, Berkeley. He consults widely with computer and electronics companies. He holds a B.S. (1971) in electrical engineering from Massachusetts Institute of Technology in Cambridge, and an M.S. (1973) and Ph.D. (1974) in computer science from Stanford University in California.