Design of CPU Cache Memories[†]

Alan Jay Smith
Computer Science Division, EECS Department
University of California
Berkeley, California 94720, USA

Abstract

We present an overview of the current issues in the design of CPU cache memories. Our stress is on those issues of greatest concern to cache designers and builders, including line size, associativity, real vs. virtual addressing, main memory update algorithm, split (data / instructions) cache vs. unified cache, cache consistency mechanisms, cache size and number of cache levels. Brief mention is made of other aspects of cache and S-unit design. The Fairchild CLIPPER^{tm} is used as an example of modern cache memory design.

†The material presented here is partially based on research supported in part by the National Science Foundation under grant DCR-8202591, and by the State of California under the MICRO program.

Proc. IEEE TENCON, Seoul, Korea, August, 1987.

Design of CPU Cache Memories†

Alan Jay Smith
Computer Science Division, EECS Department
University of California
Berkeley, California 94720, USA

Abstract

We present an overview of the current issues in the design of CPU cache memories. Our stress is on those issues of greatest concern to cache designers and builders, including line size, associativity, real vs. virtual addressing, main memory update algorithm, split (data / instructions) cache vs. unified cache, cache consistency mechanisms, cache size and number of cache levels. Brief mention is made of other aspects of cache and S-unit design. The Fairchild CLIPPER™ is used as an example of modern cache memory design.

1. Introduction

1.1. Cache Definition

Caches are high speed buffers that are used to hold items in current use. They appear in CPUs as buffers for main memory, in main memory and disk controllers as buffers for the disk address space, and in the form of your desk top as a buffer for your file cabinet. Caching is an extraordinarily powerful concept, since it can be used whenever there are uneven levels of (re)use for items or information; the 80/20 rule or Zipf's law [Knut73] are other expressions of this uneven use.

In CPUs, cache memories are high speed (associatively addressed) buffer memories that are used to hold a (time varying) portion of the main memory contents. Most computer designs can be partitioned as is illustrated in figure 1, in which the machine is shown to consist of an I-unit (instruction fetch and decode), E-unit (execution), S-unit (storage), C-unit (channel, or I/O controller), and main memory. The cache memory is located in the S-unit, along with other S-unit components such as the translator, the TLB (translation lookaside buffer), the memory interface, write through buffers, address space identifier tables, etc. The term *cache memory* is sometimes used to refer only to the buffer memory previously defined, and sometimes is used to refer to most or all of the S-unit.

1.2. Cache Justification

It is extremely difficult and expensive to build a main memory in a computer that can respond to read and write requests from the I and E units quickly and frequently enough so that the machine does not become strictly limited in its performance by main memory speed. Main memories are built from logic whose principal virtue is density, not speed, and are large and physically distant from the other parts of the CPU; main memory performance is inherently limited. Cache memories, since they are much smaller, can be built with very fast (expensive and not-dense) logic and can be placed physically near the other CPU functional units. These technology factors have led to the increasing prevalence of caches in computer systems, such that now there are almost no computer systems of moderate or higher performance that lack caches. (The Cray machines do not have caches, but are designed with different (vector) workloads in mind and different cost performance constraints than most other machines, and thus the lack of caches in those machines can probably be justified for vector processing; we do suspect, however, that the addition of caches to those machines would improve their performance for scalar processing.)

Even with the inclusion of cache memories, almost all CPUs are still most strictly limited in their performance by the cache access time: in most cases, if the cache access time were decreased, the machine would speed up accordingly. Similarly, multiprocessor computers with shared memory are most strictly limited by main memory bandwidth, and the main memory traffic from each processor is determined by the success of the cache memory in satisfying memory requests without main memory opera-

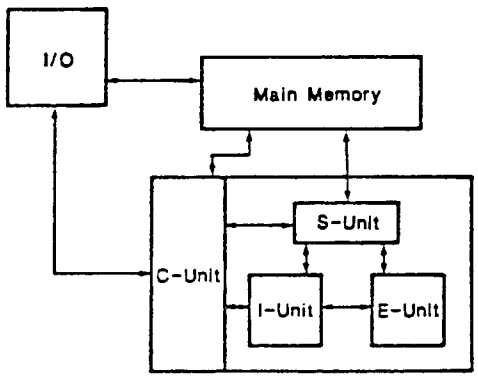


Figure 1: Computer Structure

†The material presented here is partially based on research supported in part by the National Science Foundation under grant DCR-8202591, and by the State of California under the MICRO program.

tions. Accordingly, it is widely acknowledged that *memory system performance, and in particular, the cache memory performance, is the most significant factor in achieving high machine performance.*

1.3. Why Do Caches Work?

Cache memories function effectively because of an empirical observation known as the *principle of locality*, [Denn72] which can be stated as: Information in use in the near future is likely to consist of that information in current use (*locality by time*) and that information logically adjacent to that in current use (*locality by space*). The principle of locality is widely applicable, and as noted earlier, it finds expression as the "80/20 rule" and "Zipf's law". While one can imagine reference patterns that would defeat existing cache memory designs, it is the author's experience that cache memories improve performance for any program or workload which actually does useful computation. We therefore discount the comments that begin: "yes, but for my application. ..".

1.4. Factors Determining Cache Performance

For the reasons given above, the proper design and implementation of cache memories has become increasingly important to computer designers. In order to evaluate a cache design, it is necessary to specify one or more *figures of merit*. There are two primary factors:

(1) (Mean) Access time to the cache on a hit (data in the cache).

(2) Probability of finding the referenced information in the cache.

When a reference (read, write, instruction fetch) is made to a cache, the reference can either find the needed information already in the cache (a *hit*) or a main memory operation can be required (a *miss*). The *miss ratio* is the fraction of references that miss, and the *hit ratio* is 1-miss ratio. If m is the miss ratio, Th is the time to reference the cache on a hit and Tm is the time to make a main memory reference when there is a miss, then the mean memory reference time is $T = m*Tm + Th + E$, where E is a term to account for the secondary factors discussed immediately below. Since m is usually small and Tm large, m and Th are the most important factors.

There are also a number of secondary factors affecting cache performance: (3) time to satisfy a read miss (Tm), (4) memory bandwidth generated and consequent queueing effects to main memory in a multiprocessor system, (5) cache cycles "stolen" to maintain multiprocessor memory consistency, and (6) delays due to queueing for main memory writes. We refer to these as secondary factors, since in a properly designed system, small changes in these factors have much less effect on system performance than small changes in the first two factors.

1.5. Overview

In the remainder of this paper, we will first show how a typical cache memory would look and work, and then we will discuss in more detail the specification of cache aspects and parameters, including the line (block) size, the placement algorithm, the addressing method for the cache (virtual or real addresses), the method by which main memory is updated, whether a machine should have one cache or two (data/instructions), how consistency is maintained when there are multiple processors sharing memory, and whether there should be a single level or multilevel cache. That material is followed by a very brief mention of other aspects of cache and S-unit design.

Throughout this paper, we concentrate on providing brief but clear discussions of the various issues. We refer the reader to [Smit82] and [Smit84a] for more detailed and extensive survey presentations. A very extensive bibliography of the literature appears in [Smit86]. Except when necessary, we do not cite and survey the research literature here; the interested reader should see the papers noted for the appropriate references.

2. A Typical Cache Design

Before we discuss in some detail the various components of the cache and the relevant design tradeoffs, we will consider in this section the design and operation of a "typical" cache memory, that for the Fairchild CLIPPER[tm]* [Cho86, Holl87]. We've selected this machine for discussion for three reasons: (a) we're very familiar with its design; (b) it is modern and state of the art; and (c) it is more similar to mainframe caches than to the primitive caches typically associated with microprocessors; as such it is a better example of future designs than other existing microprocessor caches.

2.1. Overview

The Fairchild CLIPPER is a new (first shipped 1986) high performance computer module consisting of three chips, a processor chip and two CAMMU (cache and memory management unit) chips, as is illustrated in figure 2. The S-unit for CLIPPER is implemented as two CAMMU chips, one each for instructions and data; the I- and E-units are both on the processor chip, which we do not

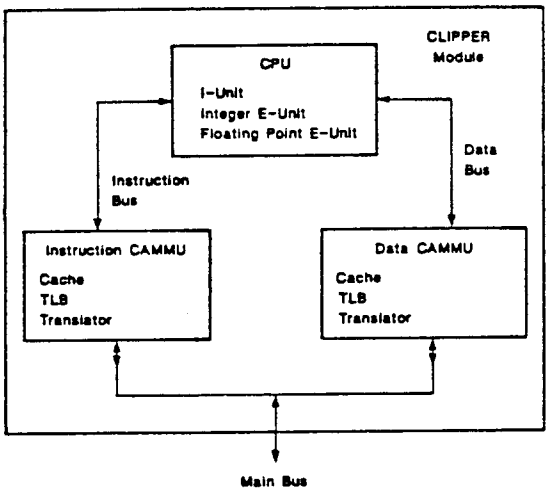


Figure 2: Fairchild CLIPPER Module

*CLIPPER is a trademark of Fairchild Semiconductor Corporation

discuss in this paper. CLIPPER is a 32-bit machine; all addresses are 32-bits, and in normal operation are virtual addresses. (A real address mode exists, but would normally be used only during startup.) The page size is 4096 bytes. The high order 20 bits of a virtual address are the virtual page address, which are translated in a two step process through a 1024 entry page directory and a 1024 entry page table.

Figure 3 is a diagram of the CLIPPER CAMMU and figure 4 is a flowchart of the CAMMU operation; each diagram contains both control and data flow lines, where appropriate. Each CAMMU contains a 128 entry, 2-way set associative TLB, a 4096 byte cache, consisting of 256 16-byte lines, organized also 2-way set associative, and a translator. We review the CAMMU operation on a CPU read here, although we don't discuss all of the features shown in figures 3 and 4.

2.2. Cache Operation

The 32-bit virtual address for a memory reference (data read or write, instruction fetch) is received over the bi-directional address and data bus between the CPU and each CAMMU chip. Parts of that address are named in various ways, as is shown in figure 5; we note that one partition splits the address into the page number, the line number and the byte within line. (Note the bit numbering in that figure.) The page number is directed to the TLB for translation. The low order 6 bits of the page number are used to select one of 64 pairs of entries in the TLB (each pair is known as a "set"). Each TLB entry contains a number of fields, including the virtual and real addresses of a page, and various control and protection flags. The virtual address field of each of the two entries selected is compared to the remaining 14 high order bits of the actual virtual address; if there is a match, then the corresponding real address is compared to the (real) address field of the cache lines selected, as is explained below. If neither virtual address from the TLB matches the one from the CPU, then the translator translates the virtual address, via the page directory and page table, to a real address, and loads the new mapping into the TLB, replacing the least recently used entry in the set. The TLB is again accessed, continuing as above; path "D" in figure 4 shows a possible optimization not implemented in the current version of CLIPPER.

The low order 7 bits of the line number (of the 8 bits available) are used to select one of 128 sets of two entries in the cache. Each cache entry consists of 16 bytes of data (a line or "quadword"), a real address tag, and some control bits (valid, dirty). The real address tag (bit 11, and bits 12-31 of the real address) of each of the pair of entries is read into a comparator and compared with the real address from the TLB (along with bit 11 of the virtual address). Simultaneously, the two corresponding quadwords are gated through multiplexors which use the word within line bits to select the addressed word. If the addresses match, then the corresponding word is selected via another MUX and then forwarded to the CPU over the CPU/CAMMU bus. If there is no match between the real address (from the TLB) and the real address tags from the two lines in the cache, then the real address is sent to main memory, main memory responds with the contents (16 bytes) of the line corresponding to that real address, the LRU entry in the cache set is loaded with that line, and the access proceeds as above. If the entry to be replaced is "dirty", i.e. modified with respect to main memory, it must be copied back to main memory prior to the fetch of the new line. (Note that path "E" in figure 4 shows a possible optimization not implemented in the current version of CLIPPER.)

For both the TLB and the cache, after a match is detected, the LRU bit for the set is set to reflect which entry is least recently used.

One very effective "short cut" to the above operation is also used. When the address is received from the CPU, it is compared to a register which has the (virtual) address of the contents of the quadword most recently referenced. If there is a match, then this reference is satisfied by selecting the next word from the corresponding quadword buffer, and no cache or TLB access takes place, thus saving considerable time. (See path "C" in figure 4.)

The above sequence of operations is quite similar to that of the Amdahl 470, IBM 370/168 or the DEC VAX 11/780 [Smit82]. In comparison to cache memories for most current microprocessors, it is considerably more sophisticated.

3. Design Choices

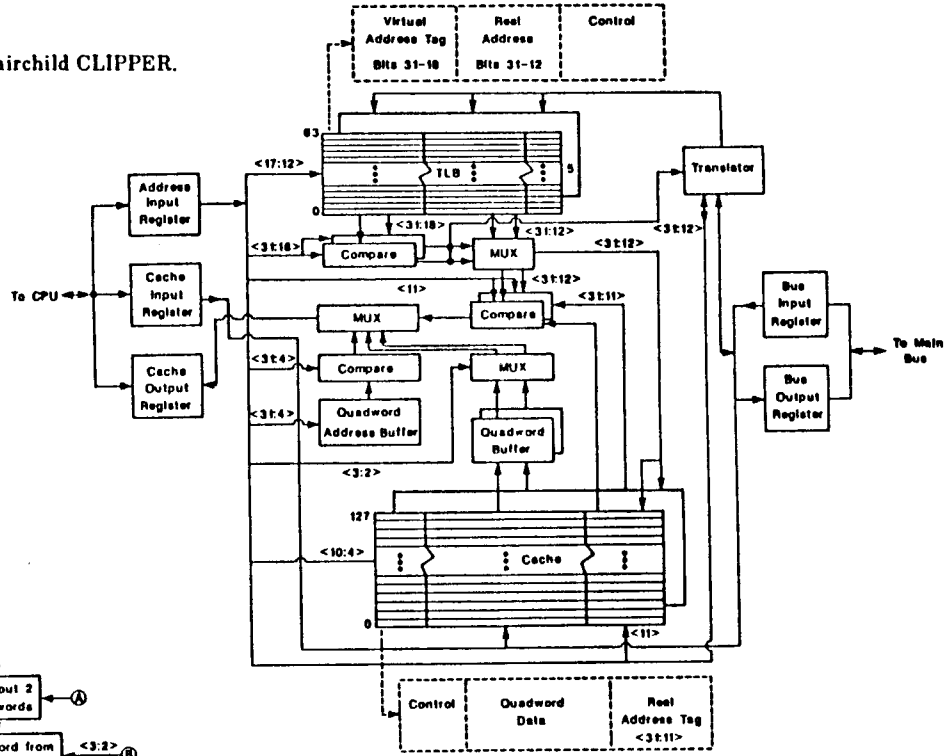
The design of the cache for the Fairchild CLIPPER, as noted, is fairly "typical" of modern cache memory designs, but we need to distinguish between "typical" and "standard"; there is no standard design. First, the cache design is sensitive to price/performance considerations, so that choices appropriate for a single chip microprocessor are often not suitable for a high end mainframe. Second, the design is sensitive to changes in technology, and in particular to factors such as memory access time, cache access time, chip density, bus speed, the ability to include a cache on the CPU chip, etc. The choice of design may be affected by the extent to which the designers are willing to consider more complex and higher performance designs, which is in turn affected by the availability of suitable CAD tools and enough time before promised product shipment. Finally, within the space of acceptable (if not optimal) designs, there is still considerable latitude for variation.

In this section, we discuss a number of cache parameters or design choices, giving the arguments which influence the choice, and when appropriate making some comments about where the best choices lie. We start with a discussion of line size, and then cover the cache placement algorithm (i.e. mapping and associativity), virtual vs. real addressing, the main memory update algorithm, the use of a split (data/instructions) cache, the problem of cache consistency in multiprocessor systems, and the use of multilevel caches.

3.1. Line (Block) Size

The line size (or block size, as it is sometimes called) is the unit of data transfer between the cache and main memory. Each line in the cache has an address tag associated with it. For most of this section, we assume that a line is either present or absent in the cache, although sector cache designs [Hill84] permit partial lines (*sub-sectors*) to be present.

Figure 3: Design of Cache and Memory Management Chip for Fairchild CLIPPER.



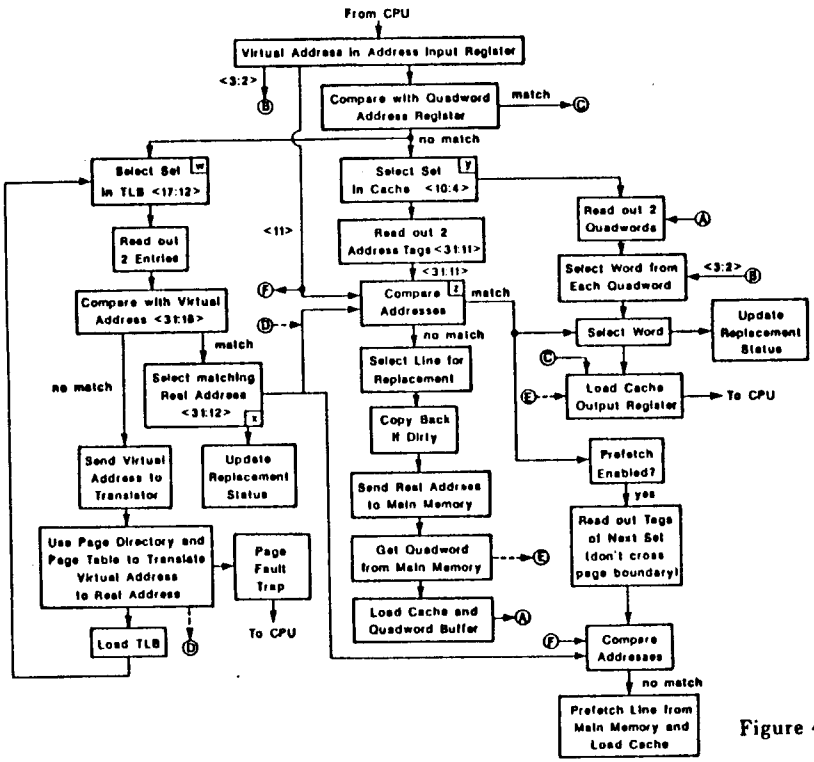


Figure 4: Operation of Cache and Memory Management Chip for Fairchild CLIPPER.

[Smit85c] "CPU Cache Consistency with Software Support and Using "One Time Identifiers"", Proc. Pacific Computer Communication Symposium, Seoul, Republic of Korea, October 22-24, 1985, pp. 142-150.

[Smit85d] Alan Jay Smith, "Disk Cache - Miss Ratio Analysis and Design Considerations", ACM Transactions on Computer Systems, 3, 3, August, 1985, pp. 161-203. (Early version available as UC Berkeley CS Report UCB/CSD83/120.)

[Smit86] Alan Jay Smith, "Bibliography and Readings on CPU Cache Memories", February, 1986. Computer Architecture News, 14, 1, January, 1986, pp. 22-42.

[Swea86] Paul Sweazey and Alan Jay Smith, "A Class of Compatible Cache Consistency Protocols and Their Support by the IEEE Futurebus", Proc. 13'th Ann. Int. Symp. on Computer Arch., Tokyo, Japan, June, 1986, pp. 414-423.

[Tang76] C. K. Tang, "Cache System Design in the Tightly Coupled Multiprocessor System", Proc. NCC, 1976, pp. 749-753.

Changes in the line size affect a number of aspects of the system design and performance. First, the miss ratio will change. As the line size increases from very small to very large, the miss ratio will initially decrease, since a miss will fetch more data at a time. Further increases will then cause the miss ratio to increase, as the probability of using the newly fetched (additional) information becomes less than the probability of reusing the data that is replaced.

To determine the effect on performance, miss ratio must be combined with the time to service a miss. That time is typically of the form $aZ+b$, where Z is the line size (in bytes), a is the marginal transfer time per byte and b is the overhead per miss. Typical values for b for a system based on a common bus (see figure 7) might be 100ns to 400ns, and for a might be 10ns to 25ns (40ns to 100ns per bus cycle, for a 32 bit bus). Thus, typically, $b >> a$.

The issue of selecting an optimal line size is considered in detail in [Smit85b]. In table 1, we show the range of line sizes for which the mean main memory delay per instruction is (approximately) minimized, for three values of a and b for a unified cache; corresponding data for instruction and data caches appears in [Smit85b]. For CLIPPER, the line size chosen was 16 bytes, within the optimal range of 16-64 bytes for the instruction cache and 8-64 bytes for the data cache. (Other factors, noted below, also figured in the line size choice.)

There are a number of other factors which also influence the choice of line size. The size of the line affects the fraction of the data storage in the cache that is actually allocated to main memory data. For example, if the address tags for a line take two bytes (16 bits) and the lines are 4 bytes, then only 2/3 of the data is "useful"; if tags are 2 bytes, and lines are 64 bytes, then only 1/33'rd of the data storage is "wasted" by tags. This means that increases in the line size actually slightly increase the effective cache size.

Another factor is the frequency of *page crossers* and *line crossers*. A page crosser is a memory reference spanning the boundary between pages, and a line crosser spans the boundary between cache lines. (Neither is possible in the CLIPPER, which requires aligned accesses, but both

Optimal Line Sizes (Bytes) - Unified Cache

Cache Size (bytes)	a=15ns/byte b=360ns	a=15ns/byte b=160ns	a=4ns/byte b=600ns
32	4-16	4-8	8-16
64	8-16	4-16	8-32
128	8-16	4-16	8-32
256	8-32	8-16	16-32
512	8-32	8-16	16-64
1024	8-32	8-16	16-64
2048	16-32	8-32	16-128
4096	16-64	8-32	32-128
8192	16-64	8-64	>=64
16384	16-128	8-128	>=64
32768	16-128	8-128	>=64

Miss Service Time = a*(line size) + b
Table 1

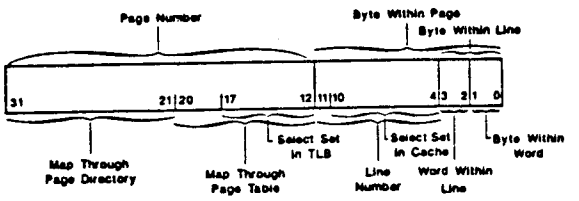


Figure 5: Interpretation of Virtual Address.

are possible in the IBM 370 architecture.) Typically, a line crosser reference will take additional time, since it is usually implemented as two accesses. One of the most difficult aspects of CPU implementation is handling exception conditions, such as traps, interrupts and page faults. A page crosser not only suffers from the same penalties as a line crosser (since pages are multiples of a line size), but can potentially result in up to two page faults, one on each page touched; the CPU or cache complexity increases accordingly.

As the lines get longer, a miss ties up the memory interface for a longer period of time. This can affect performance in a multiprocessor system, where other processors may be locked out of shared memory for longer periods. It can also cause *I/O Overruns*, which are events in which an I/O device overruns its own small I/O buffer, because it cannot reach main memory. An I/O overrun is usually handled by aborting and retrying the operation.

We briefly note the possibility of *sector caches*. A sector cache consists of *sectors* (blocks or lines) and *sub-sectors* (sub-blocks). An address tag is associated with each sector, and a validity bit with each sub-sector. On a miss, only one sub-sector is loaded, causing an entire sector to be replaced if necessary. A sector cache allows one to decrease memory traffic at the cost of an increased miss ratio; this is useful when b (the miss overhead) is not much larger than a (the transfer time per byte), but not otherwise. The behavior of sector caches is studied in [Hill84].

3.2. Placement Algorithm and Associativity

As noted earlier, caches are buffers for main memory, and hold only a (constantly changing) portion of their contents. That implies that the cache must be referenced in an associative or partially associative manner; i.e. one or more lines in the cache are retrieved, and then it is determined if any of them are the target. The degree of associativity and type of mapping affect performance.

Most caches are *set associative*, [Smit78a] which means that an address is mapped into a *set*, and then an associative search is made of that set; see figure 6. If there is only one set (S=1), then the cache is called *fully associative*, and if there is only one element per set (E=1), then the cache is called *direct mapped*. The tradeoffs involved in selecting the degree of associativity are discussed in detail in [Hill87]; we summarize them here.

Empirically, and as one would expect, increasing the degree of associativity decreases the miss ratio. We note that this is an empirical phenomenon, since one could

easily write a program which would produce the opposite effect. The highest miss ratios are observed for direct mapping. Two way set associativity is significantly better, four way is slightly better still, and further increases in associativity have little further effect. We note that changes in miss ratio with associativity are sensitive to the workload, block size and cache size, and other parameters.

Unfortunately, increasing the degree of associativity has some disadvantages. The normal parallel implementation of a cache requires that the number of comparators and data readout paths equal the degree of associativity, which is expensive, or if implemented on one chip, requires significant extra silicon area. The output of the comparisons are fed together (e.g. in a wired-or configuration or into a multiplexor controlled by the comparator outputs), and the speed of that circuit worsens with increased fan-in. A direct mapped cache can be even faster, since the comparison requires only a "go-no go" decision, with no fan-in at all. (The go/no-go decision can also take place late in the cycle and still successfully abort the operation if negative.) The optimal degree of associativity is a function of the miss ratio, since small changes in an already low miss ratio are unimportant relative to small changes in the mean cache access time, and conversely. Using reasonable assumptions about circuit technology and main memory access time, in [Hill87] results are presented suggesting that for caches larger than about 8-32Kbytes, direct mapping gives equal or slightly better performance than 2-4 way set associativity, and for smaller caches, a set associative design is better.

Another factor affecting the degree of associativity in a cache accessed with real addresses is the desire to overlap the cache access with the address translation through the TLB. The overlap occurs because the path in Figure 4 from the boxes labeled "w" through "x" can be overlapped with the path from "y" to "z". If the page size is 4Kbytes (as above), then the available untranslated address bits can select among at most 4Kbytes. If a 16Kbyte cache is needed, then four way associativity is required if translation is to be overlapped. This is the reason that the IBM 3033 (with a 64Kbyte cache) has 16-way associativity. We note that in some machines, there is a small, on-chip TLB, and a large off-chip cache; in that case, overlap is difficult or infeasible, and all real address bits are available "immediately" for cache access.

The considerations above suggest that degrees of associativity of 1 to 4 are best, unless there is a need to overlap translation and access in a very large cache.

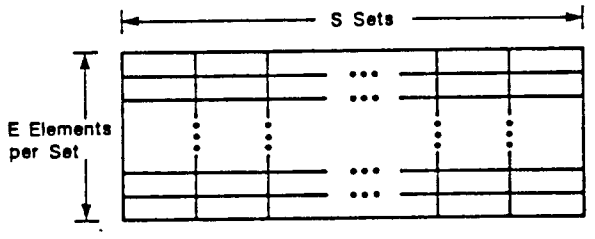


Figure 6: Set Associative Mapping

In the description of the operation of the Fairchild CLIPPER, we noted that the set was selected using 7 bits to select one of 128 sets. This is the simplest mechanism, and is called *bit selection*. It is also possible to randomize or hash some number of bits in order to select the set, but as is shown in [Smit82], this yields no advantage in miss ratio, and adds complexity and access time.

3.3. Virtual vs. Real Addressing

Portions of a program's address space can be identified two ways - by their virtual address, and by their real address. The virtual address is that generated by the program; after that address is translated by reference to the page table, it becomes a real address and then refers to a specific physical main memory location. This translation process is invisible to the program, so that from the program's point of view, there is no significant difference between the two.

An examination of the discussions above in sections 2 and 3.2 suggest that designing a cache memory to be referenced on virtual address would be advantageous. First, since no translation is required (partially overlapped or not), access to the cache should be faster. Translations to real addresses (for reference to main memory) can be done only when needed, need not be done quickly, and in any case, can be overlapped completely with the initial cache access. Second, because there is no need to attempt to overlap translation and cache access, the degree of associativity in the cache can be set at will, and is otherwise unconstrained. Although all address spaces have the same set of virtual addresses within them, by attaching an address space identifier to all address tags, virtual addresses can be made unique. Due to the synonym problem (described below), however, virtual caches are difficult to build.

The existence of virtual memory and the real to virtual mapping means that it is possible that more than one virtual address can be translated to refer to the same real address. This hypothetical event can occur under several circumstances: (a) Two different programs on the same CPU share some number of (real) pages and those pages are placed in different places in each program's address map. (One of those programs can be the operating system.) (b) One program has, via a call to the operating system, asked that different virtual addresses within its own (single) address space be equivalenced to the same real address. (c) An I/O device, using real addresses, references a region of main memory which is also accessible to a program via its virtual address space. (d) A program running on another CPU and sharing the same physical memory has some part of its virtual address space mapped onto the same real memory locations as the first program.

When two virtual addresses refer to the same physical address, they are referred to as *synonyms*, and the associated problem is called the *synonym problem*. The difficulty is that is that any reference to a given real memory location must affect all copies of the data in that location simultaneously and in the same way (but see our discussion below of cache consistency). As a practical matter, this has two implications for how one would build a cache referenced by virtual address (or a "virtual address cache"). (1) A given real address should be represented

under at most one virtual address in a cache at any given time. This implies that on a miss, the cache must be searched to see if the contents of the given location are already cache resident under a different "name" (virtual address); if so, the "name" of the data must be changed. This in turn requires that the cache contain a mechanism to map real addresses back to virtual addresses, for all virtual addresses resident in the cache; we call this mapping mechanism the *RTB*, or *reverse translation buffer*. (2) As discussed in more detail in section 3.6, all references to main memory must be translated to real addresses, and then retranslated by the RTB in each processor to ensure that the reference doesn't affect something resident in the local cache.

For various reasons (discussed in somewhat more detail in [Smit82,84a]), RTBs are difficult to build and use. The result is that the only commercial machine (as far as the author knows), running software which permits synonyms and containing a virtual address cache is the Amdahl 580, despite some significant advantages to such a design. We expect, however, that over time, as logic becomes cheaper and designers more experienced, virtual address caches will become more popular.

3.4. Main Memory Update

Cache memory is only a temporary buffer for main (addressable) memory, and any writes to memory by the CPU must appear in main memory either immediately or eventually. There are two basic approaches to updating main memory: *write-through* (store through), and *copy-back*. With write-through, all writes are immediately transmitted directly to main memory; when using copy-back, (most) writes are written to the cache, and then are copied back to main memory as those lines are replaced. There are a number of factors affecting the choice between these two possibilities.

Write-through has in the past been the most common approach for a number of reasons. First, it is somewhat simpler because on a replacement, it is never necessary to copy anything back to main memory. Second, it makes it somewhat easier to maintain consistency, since main memory is always up to date; with copy-back, the only valid copy of a given datum may be in a cache memory. Third, reliability is often higher, since main memory is usually implemented with elaborate error detection and correction mechanisms, whereas cache memories seldom have error detection, let alone error correction.

The main disadvantage to write-through is that it usually generates much more memory traffic than copy-back and thus can negatively affect performance in a system in which memory bandwidth is a bottleneck. (Note that if the cache is small and the line size large, copy-back can actually have higher memory traffic.) Write through can also result in low performance if there is insufficient write buffering to avoid frequent queueing on writes [Smit78]; many writes in a row are common. Write buffers can be used to avoid this problem, but their implementation is complex, since any memory reference (from either this CPU, another CPU or by I/O) must also be compared to the addresses of pending writes, to ensure that the most recent copy of information is referenced.

Because of the trend toward shared memory multiprocessors, and the resulting memory bandwidth bottleneck, copy-back caches are becoming the preferred design, but it isn't that difficult to implement both copy-back and write through. The Fairchild CLIPPER has two bits associated with each page (stored in the page table and buffered in the TLB), which specify write through, copy back or non-cacheable. Unpublished experiments show that performance is from 1% to 10% lower when using write-through than using copy-back.

We note that there is one important case where write-through is still the best design That case is when a microprocessor has a small on-chip cache, and when it is designed to also run with a larger, off-chip cache. In that case, memory consistency (see section 3.6) is most easily maintained by making the on-chip cache write-through; there is little if any performance loss, since the off-chip cache only connects directly to one on-chip cache.

There are some variants of write through and copy back caches. A cache is *write-allocate* if a write miss causes space for the line to be allocated in the cache and the line fetched. It is possible to build either a copy-back or write-through cache as either write-allocate or not-write-allocate. *Write-update* means that on a write the line is updated in the cache if present; the alternative (in a write-through cache) is *write-purge*, by which a write causes the line to be flushed from the cache.

3.5. Split Data/Instruction Cache

Traditionally, computers have been built with one (unified) cache memory; all references to memory, whether reads, writes or instruction fetches. were handled by the same cache. This is the semantically simplest arrangement, since there is only one main memory, and it ensures consistency by maintaining only one copy of information which is read both by the I-unit and E-unit, and is written to by the E-unit. Sharing the cache also leads to the most efficient use of a limited resource and thus lower miss ratios than a split instruction/data cache.

There are some disadvantages to a unified cache. Since in a highly pipelined machine, instruction fetches and data reads and writes are largely independent, access conflicts to the cache can lower performance. Second, even when there is no conflict, arbitration time may increase the cache access time. Third, partitioning the cache into separate data and instruction caches permits placing each cache physically adjacent to the corresponding functional unit (i.e. I-unit, E-unit), which cuts access time.

Until recently, the need to maintain cache consistency, while permitting writes into the instruction stream, has led to the continued prevalence of the unified cache. More recently, in some modern designs such as CLIPPER, the cache has been split to obtain the performance advantages enumerated immediately above. Consistency problems are avoided since for new architectures, writes into the instruction stream can be prohibited as a matter of architectural definition; this is not possible for new (and upwardly compatible) implementations of old architectures such as the IBM 360.

3.6. Cache Consistency

Increasing VLSI densities have recently made small computers, such as those based on high-end microprocessors, much more cost effective per instruction executed than large mainframes [Smit84b]. Such machines can be combined to provide high aggregate (as opposed to serial) performance. The programming of algorithms on such processors is generally much easier when processors share a common memory. The problem with such designs is that when the individual processors have cache memories, the cache memories can potentially hold inconsistent versions of shared data.

A number of means have been invented to maintain a consistent memory image in a shared memory multiprocessor. An overview of these methods appears in [Smit85c]; we provide a briefer summary here.

3.6.1. Shared Cache

The simplest possible solution to the cache consistency problem is to have only one cache, and to share it; just this solution was used in the Amdahl 470, in which I/O was routed through the cache [Smit78b]. There are two factors which make a *shared cache* design poor. First, the access time to the cache is the limiting factor in the machine performance for many or most high performance machines, and the access time to a shared cache would have to be greater than for a dedicated cache due to longer (physical) access paths, arbitration delays and access conflicts. Second, it is difficult to make the bandwidth of a shared cache sufficient to support even two high performance processors.

It is possible to build a machine in which each processor has its own cache for data local to that processor, and then there is another cache which is shared among all processors. Data tagged as shared would be allocated to the shared cache. This *shared/private cache* design is feasible although rather inelegant due to the lack of symmetry of reference, but does have some problems. First, it must be possible to identify what data is shared. There are a number of cases, however, (see below) and it is easy to designate far too much as shared. Second, the bandwidth of the shared cache still limits the number of processors. Finally, the slower access time to the shared cache may still affect system performance.

3.6.2. Broadcast Stores

A solution used in some machines (e.g. the IBM 370/168, 3033) is to *broadcast all stores* to all machines. The receiving machine then checks its own cache and if the target of the store is found there, then either the target is updated or (as with the 370/168 and 3033) it is invalidated.

The problem with this design is that the fraction of the cache bandwidth required to service external updates/invalidates grows (almost) linearly with the number of processors and the resulting memory interference makes this solution implausible for more than 2 to 4 high performance processors. By filtering out repeated requests to invalidate the same block, by using a BIAS Filter Memory (BFM) [Bean79], the number of processors can be increased, but then the BFM bandwidth becomes the limiting factor.

3.6.3. Centralized Directory

The first hardware based consistency mechanism to be described in the literature was that of a *centralized directory* [Tang76]. In this architecture, main memory maintains a directory which keeps track of which lines are in which caches, and in what status (e.g. shared / exclusive). When a processor requests a line for shared access, the main memory controller ensures that no other processor has that line for exclusive access by searching its directory and requiring that any processor holding the line for exclusive access relinquish it. When a processor requests a line for exclusive access (or converts a line from shared to exclusive) the controller invalidates the line in the caches of any processor holding it.

There are some limitations to the centralized directory method. First, the central controller is expensive and complicated. Second, processing of misses can be slow due to the need to reference the directory and perhaps recall lines from other caches; queueing delays will slow up memory access even further. Finally, the bandwidth of the controller must be sufficient to accommodate all processors at all plausible miss rates.

3.6.4. Common Bus Methods

With the availability of high performance, large address space microprocessors, multiprocessor designs in which several microprocessors all share the same (common) main memory bus have become popular; this architecture is illustrated in Figure 7. Because all processors share the same path to main memory, each can monitor misses from all other caches, and the directory method can be implemented straightforwardly in a distributed manner. Each cache maintains a directory for the information that is locally resident.

In general, *bus methods* for maintaining multiprocessor consistency function as follows. All caches have an intelligent interface between the cache and the common bus to main memory. Any processor operation (such as a read miss, a write miss, or an attempt to write a line which is currently unmodified) which threatens to produce an inconsistency must in some way be announced on the bus. Correspondingly, every cache interface must watch the bus, and must take whatever actions are necessary (e.g. invalidate local copy or preempt read from memory and supply local copy) to preserve consistency.

The first bus consistency method was proposed in [Good83]. Improvements and extensions have appeared since. A recent paper in this area is [Swea86], in which a class (MOESI) of compatible protocols is described. That class of protocols has the property that a processor can implement any protocol within the class (and may implement different protocols for different pages, or may change the protocol over time), and still maintain consistency even though other processors may implement different (compatible) protocols. The MOESI class of protocols includes a number of the previously published ones as special cases.

There are several problems with the bus methods which limit their applicability. First, all processors must share a common bus, which may be difficult due to aggregate memory traffic and other reasons such as bus length, bus loading, physical configuration, etc. Second, the bus traffic limitation sets an upper limit on the number of pro-

cessors that can be accommodated. Finally, the interface between the cache and the bus must be fairly sophisticated.

3.6.5. Software Methods

In order for a computation to be deterministic, synchronized access to shared resources, such as memory, must be enforced. This implies (although not trivially) that the operating system software knows which areas of memory will be shared and when, and it can issue commands to the various processor caches so that references to that shared memory will be correct. Therefore, it is also possible to implement consistency via software.

There are two issues in enforcing consistency: (1) When a processor requests a line for a read, it must get the latest value; this can be arranged if all misses are serviced from main memory and if main memory is itself guaranteed to be up to date. Main memory can be made current either via write through, or by flushing modified data from a cache when the shared region is released. (2) When an area of memory is referenced sequentially by processors A, then B, and then A again, we must be sure that the values that A sees are not "stale", through having remaining in A's cache while B modified them. The traditional solution to this problem is to have A purge its own cache when it releases the shared area of memory, although in [Smit85c], a means is described to prevent access to shared data without purging the cache. (A "unique identifier" is attached to both the line and the corresponding TLB entry, and must match for access to be permitted. Deleting that identifier prevents further access.)

Software enforcement of cache consistency has one general problem, which is that it must be possible to identify just when cache must be purged or flushed. While this is presumably possible, since correct operation requires correct synchronization, performance considerations make this very tricky. The problem is that synchronization operations are much faster (less costly) than cache flush or purge operations. Consequently, if the cache is flushed or purged on every synchronization operation, performance will be very adversely affected; if the cache is not, then programming errors are likely to be introduced, and debugging is likely to be difficult.

The trend is toward the use of bus based consistency methods for systems with small or moderate numbers of processors, and toward software control for systems with very large numbers of processors. Cache consistency is a very active area of research and hopefully better solutions than those described above will be developed in the next few years.

3.7. Cache Size and Multilevel Cache

The size of the cache is one of the two parameters (along with line size) which most strongly affects the miss ratio, but cache size is usually constrained by a variety of factors. Clearly, the larger the cache, the lower the miss ratio and all other things being equal, the better the cache performance. Larger caches, however, tend to be slightly slower even when implemented in the same technology and in the same place (board, chip) in the system, due to larger fan in/ fan out requirements on circuits and slower circuit rise times. Further, cache sizes are constrained by physical limits such as chip and board area, economic considerations such as cost, and related limits such as power and cooling.

As an aid to selecting a *cache size*, keeping in mind the above tradeoffs and constraints, we have developed [Smit85a,b] what we call *design target miss ratios*, which

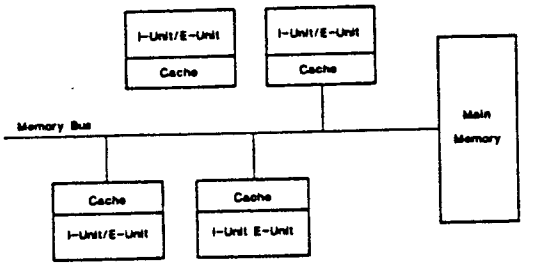


Figure 7: Common Bus System Design

are "expected" miss ratios as a function of a given cache size and line size; our design target miss ratios from [Smit85b] are shown for a unified (combined instruction/data) cache in table 2. Miss ratios to be observed in practice will vary sharply with workload, but the design target miss ratios were created for use by designers, and as a basis for comparison between designs, given some "average" workload; these design target miss ratios are comparable to those measured on real machines running real workloads [e.g. Clar83]. Beyond the range of cache sizes shown in table 2, we note that measurements at Amdahl [Smit82] suggest that the miss ratio tends to drop roughly as C**(-.5), where C is the cache size; i.e. quadrupling the cache size halves the miss ratio.

The traditional cache is just one level in a multilevel memory hierarchy, but recent technology trends increasingly favor *multilevel caches*. Using SSI, MSI or LSI technology, caches are implemented as separate collections of chips, and it is difficult to create sufficiently different performance and cost levels such that more than one level is justified. With the ability to build an on-chip cache (Motorola 68020, 68030) or a single chip cache (Fairchild CLIPPER), a multilevel cache is much more appealing. The reason for the change is that with VLSI, different levels of cache are differentiated by both significantly different access times and also significantly different sizes, although both levels may be implemented using the same circuit technology. Thus the >50% miss ratio for the caches in the 68030 almost mandates a larger off-chip cache, and the approximately 10% miss ratio for the CLIPPER [Cho86] justifies a very large board size cache (e.g. 256K) in a high performance multiprocessor system.

Another factor affecting cache size is the choice of chip technology. Usually, one builds an off-chip cache of

Design Target Miss Ratios

Line Size:	4	8	16	32	64	128
32	0.717	0.556	0.500	0.722	0	0
64	0.686	0.488	0.400	0.410	0.672	0
128	0.674	0.467	0.350	0.330	0.400	0.630
256	0.643	0.420	0.300	0.258	0.276	0.386
512	0.596	0.390	0.270	0.216	0.197	0.257
1024	0.473	0.309	0.210	0.162	0.137	0.151
2048	0.405	0.258	0.170	0.124	0.098	0.093
4096	0.329	0.193	0.120	0.082	0.059	0.050
8192	0.232	0.135	0.080	0.050	0.033	0.025
16384	0.182	0.103	0.060	0.036	0.023	0.016
32768	0.124	0.070	0.040	0.024	0.014	0.009

Table 2
Design Target Miss Ratios, Unified Cache

the fastest chips that one can buy; these are currently faster than 5ns in ECL and about 10-12ns in CMOS. The problems with such fast chips are cost, power, and density, and one often will use somewhat less than the fastest technology; price drops rapidly with decreasing performance and density also increases. (The fastest static CMOS chips are 16K or 64K; the largest dynamic RAMs in commercial production are 1M.) This same tradeoff carries over to on-chip caches. Using the same technology as the 1Mbit DRAM, one can have approximately 32Kbytes of DRAM or 8Kbytes of SRAM on 25% of a processor chip. By using the data in table 2, and making some reasonable assumptions about the time to satisfy a cache miss and the time to reference the cache (for the two technologies), one can decide between the two. (Calculations by the author favor the DRAM design.)

We see the trend in microprocessors toward increasingly large on-chip caches, with 2'nd level board-size caches in higher performance and multiprocessor systems. The chip-size caches of the CLIPPER represent the technology available in 1986. Using that technology, on-processor-chip cache sizes are insufficient to justify their use. The CAMMUs in the CLIPPER are equivalent to the on-chip cache of the future.

3.8. Other Issues

There are many other issues in cache design besides those discussed above. For reasons of space, we only list some of those other issues here; we refer the reader to [Smit82,84a] for more extended discussions. We particularly note the issues of fetch algorithm (demand/prefetch), replacement algorithm (LRU, random, FIFO), error detection and correction, pipelined access to the cache, and arbitration for multiple ports to the cache.

3.9. Other Types of Caches

As we noted in the introduction, caches are used in many other ways and places in computer systems. Here we list a number of those other ways. We note that TLBs are small caches; see [Smit82,84a]. Instruction buffers are restricted instruction caches. The branch target buffer is a cache for branch targets [Lee84]. Disk caches cache portions of the disk address space [Smit85d].

4. Conclusions

Memory system performance is currently the most critical aspect of computer system design, and the performance of the cache memory is the single most important architectural factor in the CPU performance. In this paper, we have described cache memories, provided an example of a modern cache design, and then have discussed, in varying detail, a number of factors that need to be considered by the computer system designer.

We believe that cache memories will continue to be important. Continuing research is needed on multi-cpu cache consistency, quantifying the cache performance as a function of parameter values, the design of virtual address caches, the use of caches in vector processors, the characterization of workloads as they affect caches, measuring the behavior of real machines using hardware monitors, the development of improved prefetch algorithms, the design of caches for static workloads (such as microcode) and the design of multilevel caches.

Acknowledgments

My thanks to Howard Sachs, James Cho and Walt Hollingsworth, with whom I consulted on the design of the Fairchild CLIPPER, and to Mark Hill who provided some of the information contained here. Thanks also to the same people for their comments on a draft of this paper.

Bibliography

[Bean79] Bradford Bean, Keith Langston, Richard Partridge, Kian-Bon Sy, "BIAS Filter Memory For Filtering Out Unnecessary Interrogations of Cache Directories in a Multiprocessor System", United States Patent 4,142,234, February 27, 1979.

[Cho86] James Cho, Alan Jay Smith and Howard Sachs, "The Memory Architecture and Cache and Memory Management Unit for the Fairchild CLIPPER Processor", March, 1986, submitted for publication. UC Berkeley CS Division Technical Report UCB/CSD 86/289. (Available from Fairchild Advanced Processor Division, 4001 Miranda Ave., Palo Alto, Ca., or from Alan Smith, UC Berkeley.)

[Clar83] Douglas W. Clark, "Cache Performance in the VAX 11/780", ACM Trans. on Comp. Sys., 1, 1, Feb., 1983, pp. 24-37.

[Denn72] Peter J. Denning, "On Modeling Program Behavior", Proc. SJCC, 1972, pp. 937-944

[Good83] James R. Goodman, "Using Cache Memory to Reduce Processor-Memory Traffic", Proc. 10'th Ann. Int. Symp. on Comp. Arch., June, 1983, Stockholm, Sweden, pp. 124-131.

[Hill84] Mark Hill and Alan Jay Smith, "Experimental Evaluation of On-Chip Microprocessor Cache Memories", Proc. 11'th Ann. Symp. on Computer Architecture, June, 1984, Ann Arbor, Michigan, pp. 158-166.

[Hill87] Mark Hill, "Aspects of Cache Memory and Instruction Buffer Performance", Ph.D. Thesis, Computer Science Division, EECS Department, University of California, 1987, in preparation.

[Holl87] Walter Hollingsworth, Howard Sachs, and Alan Jay Smith, "The Fairchild CLIPPER: Instruction Set Architecture and Processor Implementation" January, 1987, submitted for publication, Computer Science Technical Report 329, UC Berkeley.

[Knut73] Donald Knuth, "The Art of Computer Programming, vol. 3, Sorting and Searching", Addison Wesley, Reading Massachusetts, 1973.

[Lee84] J. K. Lee and Alan Jay Smith, "Analysis of Branch Prediction Strategies and Branch Target Buffer Design", IEEE Computer, 17, 1, January, 1984, pp. 6-22.

[Smit78a] Alan Jay Smith, "A Comparative Study of Set Associative Memory Mapping Algorithms and Their Use for Cache and Main Memory", IEEETSE, SE-4, 2, March, 1978, pp. 121-130.

[Smit78b] Alan Jay Smith, "Sequential Program Prefetching in Memory Hierarchies", IEEE Computer, 11, 12, December, 1978, pp. 7-21.

[Smit79] Alan Jay Smith, "Characterizing the Storage Process and its Effect on the Update of Main Memory by Write-Through", JACM, 26, 1, January, 1979, pp. 6-27.

[Smit82] Alan Jay Smith, "Cache Memories", Computing Surveys, 14, 3, September, 1982, pp. 473-530.

[Smit84a] Alan Jay Smith, "CPU Cache Memories", to appear in *Handbook for Computer Designers*, ed. Flynn and Rossman.

[Smit84b] Alan Jay Smith, "Trends and Prospects in Computer System Design", part of proceedings of a Seminar on High Technology, at the Korea Institute for Industrial Economics and Technology, Seoul, Korea, June 21-22, 1984. Available as UC Berkeley CS Report UCB/CSD84/219. Verbatim transcript of speech published in "Challenges to High Technology Industries", Korea Institute for Economics and Technology, pp. 79-152.

[Smit85a] Alan Jay Smith, "Cache Evaluation and the Impact of Workload Choice", Report UCB/CSD85/229, March, 1985, Proc. 12'th International Symposium on Computer Architecture, June 17-19, 1985, Boston, Mass, pp. 64-75.

[Smit85b] Alan Jay Smith, "Line (Block) Size Selection in CPU Cache Memories", June, 1985. To appear, IEEETC. Available as UC Berkeley CS Report UCB/CSD85/239.

Continued After Figure 3