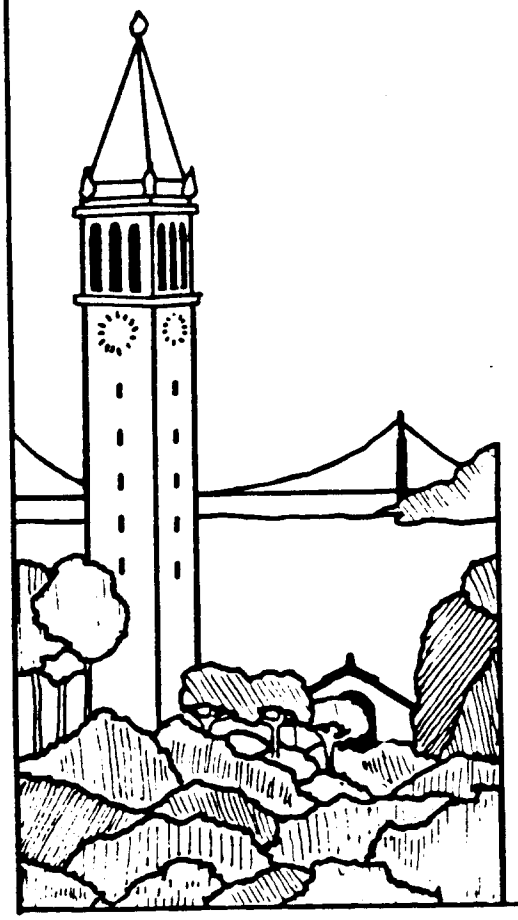
CPU Cache Consistency with Software Support and Using "One Time Identifiers"

Alan Jay Smith



Report No. UCB/CSD 86/290

April 1986

Computer Science Division (EECS)
University of California
Berkeley, California 94720

Published Proc. Pacific Computer Communications Symposium, October 21-25, 1985, Seoul, Korea, pp. 142-150.

CPU Cache Consistency with Software Support and Using "One Time Identifiers"*

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

Abstract

Multiprocessors with shared memory are currently viewed as the best way to obtain high (aggregate) performance at moderate or low cost. Shared memory is needed for the efficient and effective cooperation of processes and high performance requires the use of cache memory for each processor. A major problem is to ensure that all processors see exactly the same (consistent) view of those regions of memory that they are referencing; this is the cache consistency problem.

Almost all published and/or implemented solutions to the cache consistency problem have relied solely on hardware, and suffer from cost and performance disadvantages, especially for large numbers of processors. Some of the hardware solutions cannot be made to work at all for large numbers of processors. The generally known software solution for cache consistency requires write-through on all references and cache purging when a shared data area is released; this solution has a performance problem due to the memory bandwidth requirement of write-through.

In this paper, we propose a new software controlled cache consistency mechanism which doesn't require a shared bus and needs only limited hardware support. Shared writeable data is treated as write through in the cache; otherwise the cache is (optionally) copy-back, to minimize memory traffic. Write through ensures that the main memory copy of write-shared regions is always up to date, so that when a processor reads a line from main memory, it always gets the current value. A "one-time identifier" is associated with the TLB entry for each (shared) page and with the address tag for each line of that page that is cache resident; one time identifiers function as unique capabilities. Stale shared cache contents are made inaccessible by changing the one time identifier in the TLB entry for a page, so that the address tag on the cache line no longer matches; this avoids the need to purge the cache whenever write shared regions of memory are passed between processors. Limiting write through to data items to be read by other processors minimizes memory traffic and cache purges are required only when the supply of unique identifiers is exhausted. Our discussion in this paper also includes possible optimizations for this basic idea.

The advantages to the cache consistency mechanism proposed here include the fact that no shared bus is needed, so that memory interconnection schemes permitting much higher bandwidth are possible. This, then, permits high performance multiprocessors with shared memory to be built with many more processors than shared bus schemes allow.

1. Introduction

The development over the last few years of high performance microprocessors and high performance 'super-minicomputers' has made it very cost effective to obtain high aggregate performance by building multiprocessors [Smit84b]. The performance of the processors in this class is very sensitive to the memory access time, and can also be limited by memory bandwidth; thus cache memories [Smit82,84a] are necessary. The problem is that potentially the contents of the same word of memory can appear in several cache memories at the same time, and unless care is taken, the values for that word held in the caches can differ with each other and with main memory. This is known as the *cache consistency problem*, the problem of ensuring that any and all (legitimate) references to a word of memory at a given time obtain the same value. (We use the word "legitimate" to mean that the computation is deterministic, in the sense of the use of correct synchronization operations.)

Almost all published and/or implemented solutions to the cache consistency problem rely entirely on hardware; we survey those solutions in section 2, below. Those mechanisms have the advantage that consistency is maintained transparently and no software changes are required, but none of those mechanisms are plausible for large numbers of high performance processors: directory methods are expensive and slow, and bus and broadcast methods suffer severe performance bottlenecks for more than a small number of processors.

For large numbers of processors sharing memory, software intervention seems to be necessary to maintain cache consistency, but the standard software solution of write through caches and frequent cache purges also limits performance. The standard software solution to cache consistency functions as follows: by using write through, main memory can be guaranteed to be up to date. Between the time a write shared region is released and the time it is rereferenced by a processor, the cache must be purged, to ensure that no "stale" copies of write-shared data remain; fresh copies are then obtained from the up to date main memory. The use of write through stresses memory bandwidth, which is frequently a system bottleneck, and the purging of cache not only significantly increases the cache miss ratio, but it is difficult to implement efficiently and can be slow. The software solution to cache consistency is discussed further in section

*The material presented here is based on research supported in part by the National Science Foundation under grant DCR-8202591, and by the Defense Advance Research Projects Agency (DoD), under Arpa Order No. 4871, Monitored by Naval Electronic Systems Command under Contract No. N00039-84-C-0089.

2.6.

In this paper, we propose a new means of assuring cache consistency through the use of software control and with appropriate hardware support. *Our new mechanism works as follows: We use write through only for shared writeable data, since for read shared data, main memory is always up to date, and for nonshared data, it doesn't matter whether main memory is current. Rather than purge the entire cache in order to avoid using stale data values, we make those values inaccessible: A* **one time identifier** **(OTI)** *is appended to the TLB entry for a page and to every line within that page which is cache resident; that identifier can be considered to be a* **capability**. *When a line is loaded into the cache, the current one time identifier for that page is placed in its real address tag, and subsequent references require that the one time identifier on the line match that in the TLB entry. Those lines can be made inaccessible by changing the OTI in the TLB entry.* Generating the OTIs is discussed in section 3.4.

The two new features of our design are: *(a) Use write through only for shared writeable data, and use copy back otherwise, thus minimizing memory traffic. (b) Associate with each entry in the TLB and also with each address tag for each cache line a 'one time' identifier; the identifier in the TLB and the one in the line address tag must match in order to reference the cache. When a write-shared memory region is passed from one processor to another, the passing processor makes the shared stale data inaccessible in his own cache by destroying the existing one time identifier.*

There are several advantages to our new mechanism. Primarily, it permits shared memory designs with very large numbers of processors, since no shared bus is needed, and thus memory bandwidth limitations are much less stringent. Avoiding cache purges significantly improves performance, both by improving the cache hit ratio, and by avoiding the real time delays for that purge to occur. The selective write through keeps the memory bandwidth well below what would be needed for write through on all writes, and thus either permits larger numbers of processors to be used or permits an implementation with a lower performance and cheaper memory interconnect.

In the next section of this paper, we survey the existing mechanisms for cache consistency. (For other surveys, see [Smit82, 84a, 85a], [Yen85].) Our new design is presented in section 3, with all relevant details, and with an extensive discussion of the various optimizations possible. The last section provides a brief overview.

2. Survey of Existing Cache Consistency Mechanisms

2.1. Shared Cache

The simplest possible solution to the cache consistency problem is to have only one cache and make all the processors share it; this is illustrated in Figure 1. Just this solution was used in the Amdahl 470, in which I/O was routed through the cache; the original design for the 470 also permitted a second CPU to use the same cache, but that machine version was never built. Another type of shared cache is shown in Figure 2, where the caches are associated with the memories, and are thus shared by all processors. Analyses of aspects of a shared cache appear in [Yeh81], [Yeh83].

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; this is the reason that the multiprocessor version of the Amdahl 470 was never built.

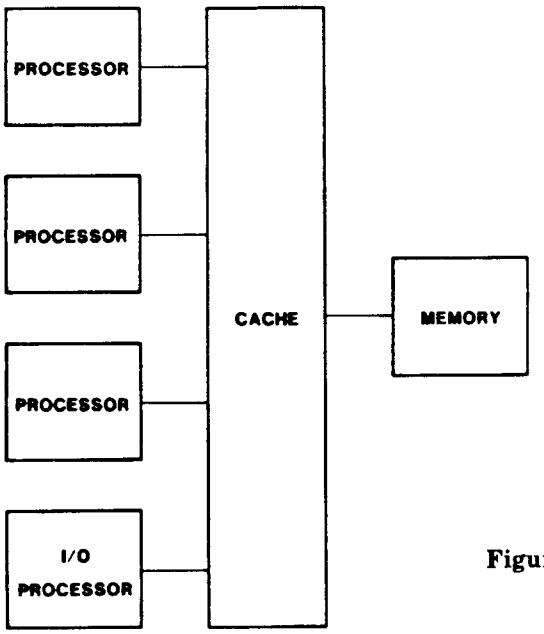


Figure 1

We do note here that our comments about how many processors a cache or bus can support implicitly assumes that those are high performance processors. If a cache or bus is made fast enough, at considerable expense, and slow (cheap) processors are used, then large numbers of processors can be accommodated. Since our interest is maximum performance at minimum cost, the latter solution is of little interest and we do not consider it further.

2.2. Shared and Private Cache

In this architecture, every 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. Such as scheme is discussed in [Flet83a,b], [IBM85], [Puza83]. Figure 3 illustrates that architecture.

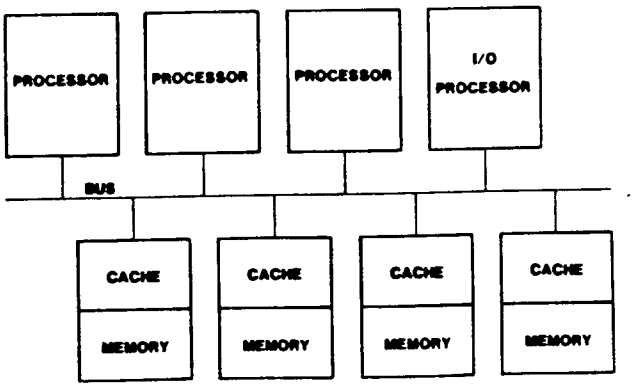


Figure 2

This 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; this is possible and our solution assumes this as well. There are a number of cases, however, (see section 3.3 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.

2.3. 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) is invalidated. This mechanism is discussed in [Jone76,77].

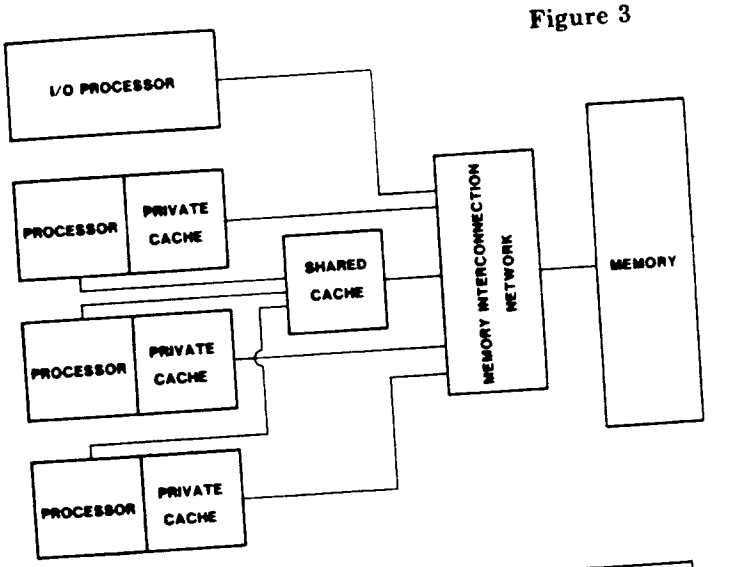
The problem with this design is that the fraction of the cache bandwidth required to service external updates/invalidates grows (almost) linearly with the number of processors and the resulting memory interference makes this solution implausible for more than 2 to 4 high performance processors, or a dozen or so medium performance processors such as the M68010. (The bandwidth required will grow somewhat less than linearly as cache access contention causes each processor to compute less rapidly and issue fewer writes.)

The use of the broadcast consistency mechanism can be extended to a considerably larger number of processors with the use of a BIAS Filter Memory (BFM) [Bean79]. The BFM remembers the **k** most recently invalidated lines and filters out repeated requests to invalidate the same cache line. The performance limit is now determined by both the bandwidth of the BFM, which has to handle all stores by all processors, and the interference with the cache by initial invalidates. If the hit ratio in the BFM were 75%, then at most four times as many processors could be used before the cache bandwidth was seriously stressed; this is still a small number of processors. (The BFM bandwidth itself might not be a problem, since it could be interleaved; the cache is generally too expensive to interleave.)

2.4. Directory Methods

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 (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. Figure 4 shows an architecture of this type, where the "memory controller" holds the directory. Analysis of and improvements to this basic scheme appear in [Arch84], [Cens78], and [Dubo82]. The IBM 3081 uses a version of this algorithm in which the System Controller (SC) maintains a copy of the directory for the cache for each CPU [Gust82]; when the SC is queried, it is called a "cross interrogate." Various

Figure 3



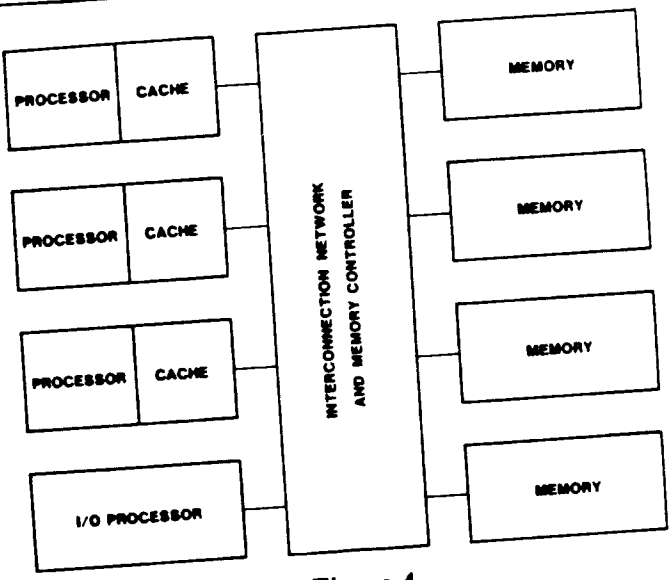


Figure 4

optimizations to the IBM design are discussed in [Bren84], [Flus83], [Hrus82], [Kryg84], and [Knig84].

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.

2.5. Bus Methods

With the availability of high performance, large address space microprocessors, multiprocessor designs in which several microprocessors all share the same main memory bus have become popular; this architecture is illustrated in Figure 5. 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. In one design, a fetch (shared access) from memory will cause any cache holding a dirty copy of the line to provide it to the requesting processor before main memory can respond. A write (exclusive access) miss, or an attempt to write to a line held as shared, will cause all other caches

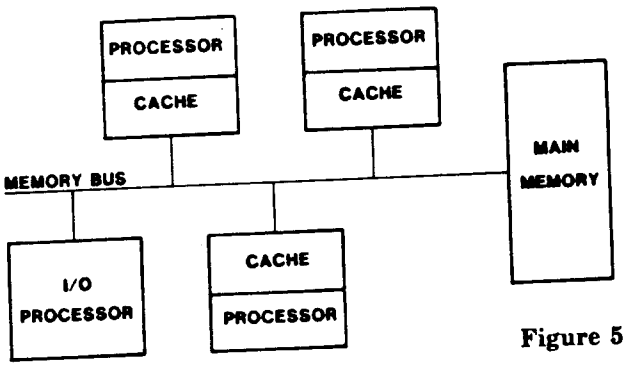


Figure 5

to invalidate their copy of the line.

The first bus consistency method was proposed in [Good83]. Improvements and extensions have appeared in [Fran84], [Papa84], [Rudo84], and [Katz85].

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

2.6. Software Enforced Consistency

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.

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. The use of write through ensures that main memory is current.

(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. This implementation is used by the RP3 [Bran85], and NYU Ultracomputer [Edle85] (both of which also make some shared information uncacheable) and is also discussed in [Maza77].

This traditional way of implementing software consistency causes two performance problems. First, all caches are write through, which burdens main memory with write traffic and may cause processors to block while waiting for writes to complete [Smit79]. Second, the frequent purging of caches when shared areas of memory are released can significantly increase the overall cache miss ratio. It is also worth noting that purging a cache which is implemented using standard RAM chips is slow; each entry has to be invalidated in turn.

3. New and Efficient Software Solution

3.1. Assumptions and Hardware Support

In order for a software consistency mechanism to work, we have to make some assumptions and have to require certain hardware support. We list those items here:

(1) We assume that memory is paged, that the processors generate virtual addresses and that the caches use real addresses. A TLB (translation lookaside buffer) [Smit82,84a] is used to translate virtual to real addresses. TLBs are one per processor, and are not shared between processors. (The real address cache is needed so that the OTI in the TLB entry can be compared with the OTI part of the line address tag. It is possible to create a design in which operation with a virtual address cache is possible; this is discussed below in section 3.5.) A TLB entry is shown in Figure 6.

(2) We assume that the relevant software (operating system or user) knows which areas of memory are write shared and when control is passed. For simplicity, we assume that access to a write shared region begins with a P(S) (semaphore request) action or its equivalent (e.g. test and set); the access terminates with a V(S) (semaphore release) action. (We assume that there is some indivisible synchronization operation such as P(S) and V(S) or test and set, or compare and swap.) We further assume that the operating system can determine which pages compose the shared region and that each shared region uses an integral number of pages. (I.e. unneeded regions of the last shared page are not allocated to something else. Our mechanism does not support a finer level of granularity of sharing than the page.)

(3) The caching is controlled on a page basis by a two bit field. The first bit (*cacheable bit*) specifies whether a line is cacheable at all, or whether it must be referenced only from main memory. (Uncacheable items would typically include semaphores (synchronization variables) and data that is write-shared with few references by each processor before access is passed. The RP3 and NYU Ultracomputer make some items uncacheable.) The second bit (*write-through bit*) specifies whether the line is to be managed copy-back or write-through. In general, as explained below, shared areas will be managed write-through. When an entry is made in the TLB, the two bit field is copied into the TLB in order to specify the appropriate type of operation on each reference (see Figure 6). We note that the settings of the write through and cacheable bits can be different for different processes, since they reside in the page table for each process. (The availability of copy back is for performance reasons only; write through is sufficient for the correctness of our mechanism.)

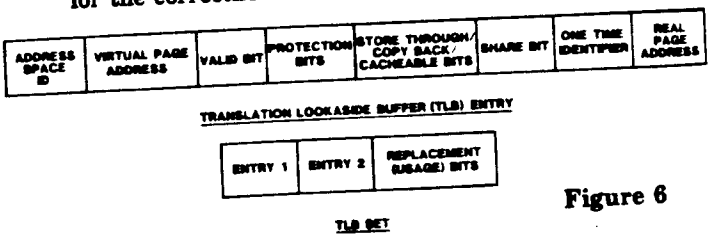


Figure 6

(4) A third bit, called the *shared bit*, specifies whether the page is shared between processors. This bit is set by the operating system, either from its own knowledge or by an explicit request from the user. For the optimized versions of the implementation of our consistency mechanism, the shared bit is not always the same as the write through bit; in the simplest implementation, it is always the same.

(5) The TLB real address field and the real address tag for each line each have room for an additional field which we call the *one time identifier* (OTI) field. When the shared bit is on, the OTI fields from the TLB and the line address tag are compared, and if they don't match, the overall match circuitry reports "no match." (If the shared bit is off, the compare does not include the OTI fields.) The function of the OTI is very similar to that of a *capability*, in that it authorizes (permits) reference. See Figures 6 and 7 for an illustration.

(6) There is an *OTI register*, the contents of which are loaded into any new TLB entry for which the shared bit is on. The OTI register is incremented every time it supplies a value. When the OTI register overflows, the entire cache is purged. There must be some mechanism to purge the cache; since it happens seldom, it need not be especially efficient. Generation of OTIs is discussed in more detail in section 3.4, where we also consider the frequency of overflow of the OTI register.

(7) There must be an *Invalidate TLB Entry (ITLBE)* command which will remove an entry from the TLB based on its virtual address. (This can also be accomplished by purging the TLB, but with significant additional performance loss. Also, as discussed below, purging all TLB entries which have the shared bit set is sufficient.)

Although the list above of assumptions and hardware requirements appears to be long, that is the result of an attempt to very clear and explicit about what is needed. In fact, the amount of hardware needed is no more than is required for any of the hardware consistency schemes, and is considerably less than for some.

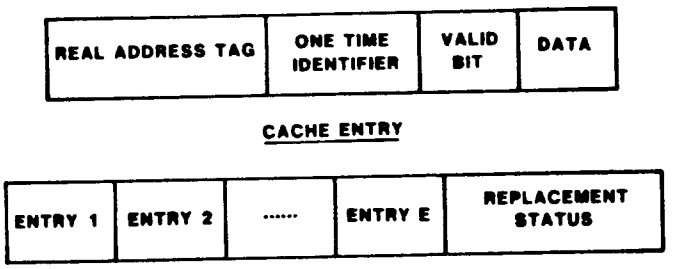


Figure 7

3.2. The (Basic) Consistency Algorithm

For the **basic version** of the consistency algorithm given here, we assume that *the shared bit is on for all pages which are write shared between two or more processors* and *any shared page is designated write through.*

Initial Use of Shared Area: The first reference to a write shared region (i.e. one with the shared bit on) or a (re)reference subsequent to the use of the region by another processor, will cause the TLB to be reloaded, automatically, via the hardware, with a new translation entry for any page referenced in that region. This is arranged (see below) by ensuring that there is no current valid TLB entry. Associated with the new TLB entry will be a new one time identifier (OTI) obtained from the OTI register.

Whenever a line is fetched which is within a page marked as shared, the OTI field from the corresponding TLB entry is loaded into the line address tag. The OTI in the TLB entry is compared on all references with that in the cache line. This will prevent access to any line which remains in the cache from previous use, i.e. stale data, since those stale entries will have a different OTI value in their tags. Any lines loaded by virtue of this new TLB entry will have the same new OTI and will be accessible via that new TLB entry.

End of Use of Shared Area: When access to a write shared region is about to be given up (equivalent to V(S)), the TLB entry for every page in that region is invalidated by issuing ITLBE (invalidate TLB entry) commands for each page. This ensures that all data in this region in the cache is now inaccessible and that subsequent references will require new fetches from main memory.

As should be clear, this algorithm avoids the stale data problem without purging the entire cache; only individual TLB entries need to be purged. Memory traffic is minimized by limiting write through to write shared regions.

In Figures 8 and 9, we illustrate the design and operation of the cache, with the special modifications we have proposed.

3.3. What is Shared, and Special Case Optimization

The issue of what is shared is not quite as trivial as it might seem on first inspection. In this section, we list all of the circumstances that lead to the same region of memory being write shared between processors. Some of those cases benefit from special optimizations. We note in particular the distinction between the use of the shared bit and the write through bit. The shared bit is to ensure that when a processor reads a region for which this bit is set, it gets fresh (up to date) data from main memory. The function of the write through bit is to ensure that when a processor with that bit set writes a region, main memory is forced to be current, and thus another processor, when reading the region currently being written to, gets the up to date values. If the use of the shared region is not symmetric, optimizations are possible, as we describe below.

3.3.1. Supervisor Data Structures

The typical shared region of memory would be one containing supervisor data structures such as the job queue. This is handled quite well using the basic algorithm.

3.3.2. Input/Output

Input/output processors (or channels) write directly to main memory, so memory is guaranteed to be current on a read. A processor reading from an input buffer must have the shared bit set for that region to be sure of getting the fresh data from memory. It need not have the write through bit set, since this is an input buffer and no other processor will subsequently read from the area.

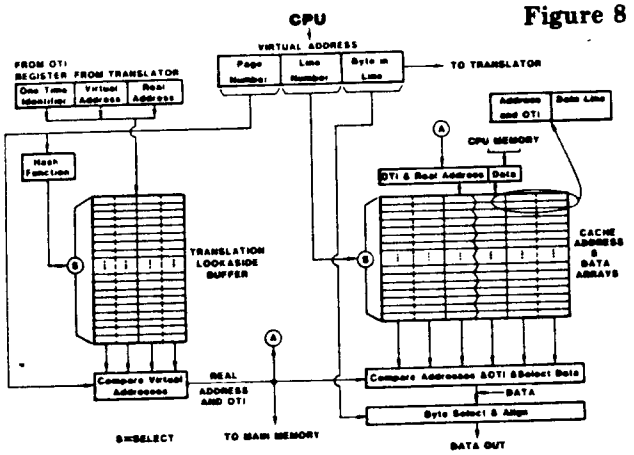


Figure 8

A processor writing to an output buffer must have the write though bit set, so that memory is made up to date. It need not have the shared bit set, since it is not reading from the region.

3.3.3. Message Buffers and Regions

In some systems, processes communicate by passing message buffers, or using mail boxes or pipes. This is one way communication, which means that the sender must use write through and the receiver must use the OTI mechanism, i.e. have the shared bit set.

3.3.4. Forking a Process

If a process is forked off, and the child processes run on the same processor, no special steps need be taken; i.e. neither the write through nor shared bits need to be set in either address space.

If a child process is to run on a different processor, then precautions must be taken. We consider two cases: when the new child has a new copy of the address space, and when the children share the existing address space and physical memory.

If a new address space copy is created, then the program (OS function) making the copy must have the write through bit set in the target pages. The child, which gets a new address space and page table, referencing the physical memory which was the target of the copy, should not have the write through bit set. It does need to have the shared bit set, since although the address space is new, the local cache may contain stale values for that region of main memory.

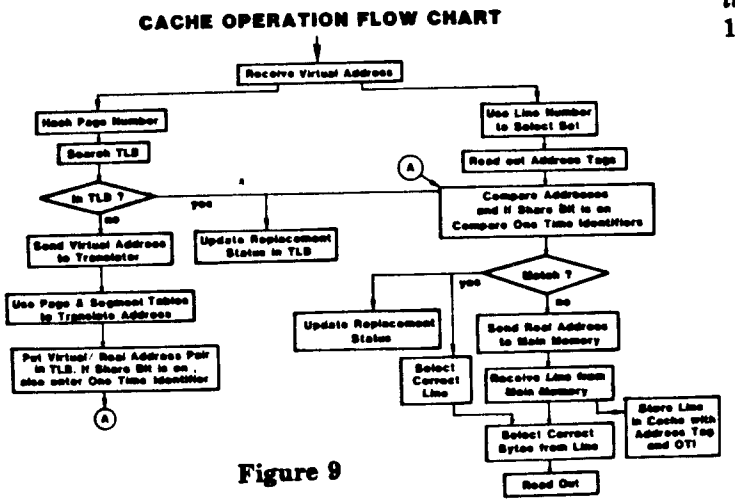


Figure 9

If the child operates in the same address space on the same (shared) physical memory, then access to data must be controlled in the same was as access to shared supervisor data structures.

3.3.5. Semaphores

Semaphores are by definition shared. (By "semaphore", we refer to those variables used for synchronization, including the targets of P(S) and V(S), and test and set instructions.) Since semaphores are only referenced on entry and exit from critical sections, they are likely to be referenced infrequently, and may be made uncacheable; if they are cacheable, they must be placed in a region which is referenced both with the write through bit and shared bit on. Further, it must be possible to operate on semaphores in an indivisible manner so as to accomplish synchronization (i.e. both the cache and main memory are locked and held until the operation is complete); this is a general requirement and is not specific to our consistency mechanism.

3.3.6. Process Migration

Ideally, in a shared memory (tightly coupled) multiprocessor system, we would like to be able to move processes from one processor to another. When memory and all I/O devices are shared between processors, this can be very easy to do, since the process is not bound to any aspect of a specific processor. (Migrating processes in a distributed system in which neither memory or I/O devices are shared is, conversely, very hard, since a process in that case is generally bound to the processor and its I/O devices.) In order for a process to be migratable, it must be treated as if it were a shared data structure: all accesses must be write through, and the processor giving up the process must purge (ITLBE) all relevant TLB entries. This solution, while correct and feasible, because of write through results in a high level of memory traffic and a consequent performance penalty or limitation.

There are two possible ways to minimize the performance penalty here of using write through. One possibility is to create a hardware cache flush mechanism, so that when the process is relinquished, the dirty lines in the cache are immediately expelled and main memory updated. This is straightforward but additional logic is required, and a considerable real time delay can elapse while the cache is flushed.

The second possibility is to create *an architecture with two levels of consistency*, such as that illustrated in Figure 10. There we see a number of busses, with several

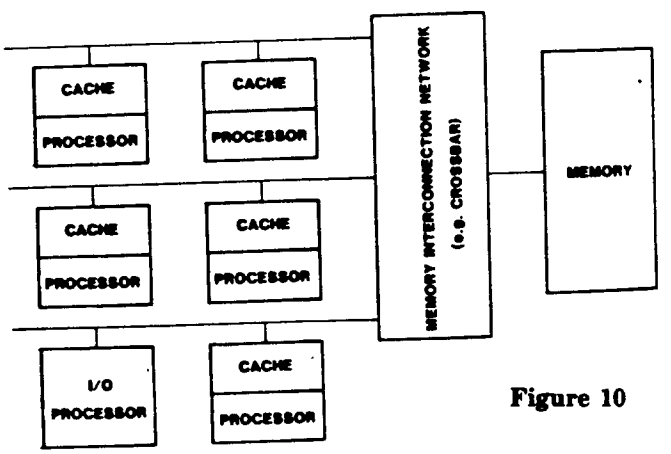


Figure 10

processors on each bus. The busses are connected via a crossbar to the main memory. In that architecture, bus-type consistency protocols can be used to ensure that all processors on the same bus have consistent caches; in that case, processes can be moved (only) among the processors on the same bus without using write through. A more extreme version of this approach is to simply forbid process migration; the dispatch queue would be particular to each processor. We believe that the advantages of a bus watch consistency mechanism are substantial, when performance considerations (i.e. bus bandwidth) permit such an architecture. Our proposed design permits the use of a shared bus for small numbers of processors, with our consistency mechanism used only for sharing between processors on different busses.

3.4. One Time Identifiers

Thus far we've given only a very cursory description of the creation and use of the one time identifiers (OTIs); we expand on that here.

The OTI consists of a k-bit field supplied by the OTI register (OTIR). The OTIR is simply a counter, which is incremented every time a value is read from it. The k-bit OTI is kept in every TLB entry and in the address tag field of every cache line. In both cases, the extra storage can be significant in relative terms, although likely not in absolute terms. Assume a machine with 16Mbytes of real storage, 32 bits of virtual address, a 4K page size and a 12 bit OTI (i.e. 4096 different OTIs available). Then a TLB entry consists of 20 bits of virtual address tag, 12 bits of real address field, and 12 bits of OTI. The real address tag on a cache line has expanded from 12 to 24 bits, including the new OTI. (With a reasonable line size [Smit85c], such as 16-bytes (128 bits), the extra 12 bits per line are less than 10% of the total storage to implement the cache.) The comparators that check the real address tags for a match have increased from 12 to 24, plus a new signal line to disable the OTI compare if the share bit is off.

The name "one time identifier" implies that every OTI value is unique and never reused, much like a capability. If it were possible to make the OTI field large enough, e.g. 48 bits, then OTIs could indeed be unique. Because of the need to conserve storage and minimize comparators, however, the size of the OTI needs to be much smaller, which means that eventually the OTIR will overflow or wraparound. If wraparound were to occur, there would be a (very small) chance that a preexisting real address/OTI pair in the TLB could be recreated, leading to a match with stale data still in the cache. The implication is that *when the OTIR overflows, the cache must be purged.*

The cache can be purged in either the foreground or background. A foreground purge means that the machine stops while a small engine (microengine or finite state machine) invalidates each address tag for each line in the cache; because the tags are typically implemented out of standard RAM chips, they can't be invalidated the way a set of flip-flops could be reset. A foreground purge thus requires a real-time halt in processing. We note that only entries with the shared bit on need to be invalidated, but every entry needs to be examined. It might also be possible to purge a cache in software, by writing a code loop guaranteed to cause every current entry in the cache to be pushed. Such a use of software simplifies the hardware, but is likely to be slower. Another difficulty with a software purge is that it must be completely clear that all existing entries are actually purged, even when traps and interrupts occur during the purge loop.

A cache purge in the background is also possible; i.e. the processor continues processing while spare cache cycles are used to perform the necessary invalidates. We add one more bit to each address tag, which we call the *bank bit* (as in "memory bank"). The bank bit is compared to the *bank flag* on each access, and the real addresses match only if the bank bit matches the bank flag. When an entry is loaded into the cache, the bank bit is always set to the current value of the bank flag. A cache purge can be temporarily accomplished by flipping the bank flag, thus causing every entry in the cache to become invalid. We say "temporarily" because every entry in the cache with the old value of the bank bit must be genuinely invalidated before the bank flag is again flipped; this can be done by an engine using spare cache cycles to invalidate entries.. If the time until the OTIR overflow is reasonably large, as would be the case for a 12 bit OTI, then all invalidations could be easily accomplished in the background. This same mechanism was used in the Amdahl 470 to purge the TLB [Smit82,84a].

It is also possible to implement the valid bits for each cache line as a set of flip flops, rather than as one bit fields within the line address tags. In that case, all of the flip-flops can be reset in one cycle with one signal.

An important issue is the size (number of bits) of the OTI. Because the OTI takes space, we would like to minimize its size. On the other hand, we would like to minimize the frequency of OTIR overflows, since they are costly both in terms of delay to purge the cache and in extra cache misses to reload the cache. As a first approximation, an OTI that is slightly larger than the log2 of the size of the TLB seems to be appropriate. For example, a 512 entry TLB with a 12 bit OTI would at most cause the cache to be purged after 2048 TLB misses, or 4 "TLB-loads". If OTIs are issued only for shared pages, then only a fraction of the TLB misses would use an OTI and overflow would be even less frequent. Since the time to overflow increases exponentially with the size of the OTI, decreasing the size of the OTI by 1 or 2 bits seems to have little payoff due to the increased frequency of overflow (and 1 or 2 bits doesn't save much storage); conversely, with an OTI of the indicated size, the frequency of overflow should be low enough that further decreases would have little merit.

3.5. Some Alternatives

For some aspects of our design, there are alternatives. In this section, we list some such which don't naturally fit elsewhere in this paper.

Instead of having the V(S) (release of shared area) function purge one's own TLB entry, it could instead broadcast a TLB entry purge (based on real rather than virtual address) and require that all other caches do a purge. This seems to be less efficient and more difficult to implement.

The ITLBE (invalidate TLB entry) command could instead be given by the P(S) function, when the shared region is acquired, rather than by V(S) when the shared area is released.

Our scheme is compatible with a *virtual address cache design* in one of two possible ways. The first way is to create a TLB that provides only the OTI as an output,

instead of both the real address and OTI; the match is then made with the OTI only. This forfeits the access time advantages of a virtual address cache [Smit82,84a]. For the second method, we have to consider how a real address cache works; see Figures 8 and 9 for an illustration. In brief, the high order bits of the virtual address are fed to the TLB which yields the corresponding bits of the real address. The middle bits of the virtual address (which are not translated, and so are the same as the corresponding real bits) are used to select the set in the set-associative cache. The real address tags from that set are then read out and compared with the now available real address. If a match is found, the appropriate portion of the line is selected and read out.

What we can do instead is as follows: we read out the tags from the chosen set and do the select based on a comparison of virtual address tags, and gate out the result if successful. Somewhat (slightly) later, the real address and OTI comparison is performed. If it is successful (i.e. yields the same result as the virtual address comparison), then no action is taken; if it is not successful, then a trap signal is sent and the instruction is restarted after the cache miss is serviced. The important point is that to do this we have to be able to halt the current instruction before it updates registers or memory.

Instead of using write through, it is also possible to have the V(S) action result in a push of all modified lines. This decreases main memory traffic at the cost, as explained above, of examining every cache entry and pushing the dirty ones. Extra hardware is needed, and every such global push causes a significant real time delay. In addition, the burst of writes over the bus or interconnection net to memory can block the other processors.

We noted above that our algorithm uses store through only for shared writeable data. It is possible to use store through for all data, at the cost of increased memory traffic, and thereby avoid the need to implement copy back as well as store through. This doesn't affect the use of the shared bit nor the OTI.

It is worth observing that the OTI mechanism provides an easy way to purge the entire cache. If all TLB entries are invalidated, then all current cache entries become inaccessible, and the cache must be reloaded. Since the TLB is substantially smaller and has fewer entries than the cache, it can be easier to invalidate than the cache itself.

3.6. The Optimized Algorithm

The optimized algorithm is exactly the same as the basic algorithm, but *the shared and write through bits are set differently. The shared bit is set only if the shared region was (or may have been) written by a different processor. The write through bit is set only if the shared region will (may) be read by another processor.* The cases for which this applies are discussed in detail in section 3.3.

These changes substantially reduce the frequency of write through and also the frequency with which OTIs are issued, and therefore the frequency of OTIR overflow and cache purge.

3.7. Costs and Performance

As noted throughout this paper, there are costs associated with our software consistency mechanism. In this section we review those costs.

Extra storage is required in the TLB and in the cache address tags for the OTIs; we estimated less than a 10% increase in cache storage and perhaps 30% increase in TLB storage, using the field sizes from section 3.4. There is also an approximate doubling of signal lines out of the TLB and address tag storage and an approximate doubling of the number of comparators required.

The engine to do a cache purge when the OTIR overflows is needed, unless the valid bits are kept in flip-flops. The additional misses caused by such a cache purge must be accommodated. The number of additional misses, with a reasonable size OTI, overall should be negligible.

The ITLBE (invalidate TLB entry) function must be provided and must be issued by the V(S) operation.

The cache should be able to support both write through and copy back (for best performance), specified on a page by page basis. The memory system must be able to support the extra traffic due to writes to the shared storage. (The amount of such traffic is very sensitive to the software and the algorithms used in the operating system. In some cases, the write through traffic would be negligible; in others, almost all stores would be write through, amounting to 10-20% of all memory references [Smit79,85b]. *This uncertainly in the frequency with which store through would have to be used prevents any detailed and exact performance analysis of the mechanism proposed in this paper.)*

The hardware required to support ITLBE, cache purge, and selectable write through/ copy back seems to be no worse than that required to implement the hardware based consistency protocols. Some extra hardware is needed, however, so a sharp reduction in hardware cost is not the primary advantage of the OTI scheme.

It is important to note that *the efficiency of our algorithm does not decrease significantly with increasing numbers of processors*, provided that the memory bandwidth is otherwise sufficient to support write through. All actions, except the store throughs to main memory, are insensitive to the number of processors. It should be possible to minimize the impact of the store throughs by very selective use of the store through feature. Because unlike the bus consistency algorithms, we can tolerate multiple data paths to memory, as with a banyan or omega network, or crossbar, additional memory traffic can be easily accommodated. (The RP3 system [Bran85] uses an omega network, and can tolerate store through on all writes, with 512 processors.)

4. Conclusions

We have described a new software controlled and hardware supported mechanism to maintain cache consistency in a shared memory (tightly coupled) multiprocessor system. It can be implemented with modest extensions to most machine architectures and with modest additional amounts of hardware to support the consistency mechanism.

The use of OTIs to maintain cache consistency provides a new and significantly improved means to high performance multiprocessing. Because our mechanism does not require a shared bus, nor a complicated central directory, very large numbers of processors can be accommodated. Since we don't require cache purges (except very infrequently) nor do we require write through

on all writes, performance should be significantly better than for other software controlled designs (e.g. [Bran85], [Edle85]). We believe that our design should have important applications for tightly coupled multiprocessor systems with large numbers of processors.

Acknowledgements

My thanks to Susan Eggers, Mark Hill, Randy Katz and Howard Sachs for their comments on a draft of this paper. Of course, any remaining errors or omissions are the responsibility of the author.

Bibliography

[Arch84] James Archibald and Jean-Loup Baer, "An Economical Solution to the Cache Coherency Problem", Proc. 11'th Ann. Symp. on Comp. Arch., June, 1984, Ann Arbor, Michigan, pp. 355-362.

[Bean79] Bradford Bean, Keith Langston, Richard Partridge, Kian-Bon Sy, Cache Directories in a Multiprocessor System", United States Patent 4,142,234, February 27, 1979.

[Bran85] W. C. Brantley, K. P. McAuliffe and J. Weiss, "RP3 Processor-Memory Element", Proc. of the 1985 Int. Conf. on Parallel Processing, to appear.

[Bren84] J. G. Brenza, "Cross-Interrogate Directory for a Real, Virtual or Combined Real/Virtual Cache", IBM Tech. Disc. Bull., 26, 11, April, 1984, pp. 6069-6070.

[Cens78] Lucien Censier and Paul Feautrier, "A New Solution to Coherence Problems in Multicache Systems", IEEETC, C-27, 12, December, 1978, pp. 1112-1118.

[Drim81] E.G. Drimak, P. F. Dutton, and W. R. Sitler, "Attached Processor Simultaneous Data Searching and Transfer Via Main Storage Controls and Intercache Transfer Controls", 24, 1A, June, 1981, pp. 26-27.

[Dubo82] Michel Dubois and Faye Briggs, "Effects of Cache Coherency in Multiprocessors", IEEETC, C-31, 11, November, 1982, pp. 1083-1099.

[Edle85] Jan Edler, Allan Gottlieb, Clyde P. Kruskal, Kevin McAuliffe, Larry Rudolph, Marc Snir, Patricia Teller and James Wilson, "Issues Related to MIMD Shared-Memory Computers: the NYU Ultracomputer Approach", Proc. 12'th Int. Symp. on Comp. Arch., Boston, June, 1985, pp. 126-135.

[Flet83a] R. P. Fletcher, R. A. Heller, and D. M. Stein, "MP-Shared Cache with Store-Through Local Caches", IBM Tech. Disc. Bull., 25, 10, March, 1983, pp. 5133-5135.

[Flet83b] R. P. Fletcher, D. M. Stein and I. Wladawsky-Berger, "MP-Shared Processor Memory", IBM Tech. Disc. Bull., 25, 10, March, 1983, pp. 5128- 5132.

[Flus83] Federick O. Flusche, Richard Gustafson, Bruce McGilvray, "Cache Storage Line Sharability Control for a Multiprocessor System", United States Patent 4,394,731, July 19, 1983.

[Fran84] Steven J. Frank, "Tightly Coupled Multiprocessor System Speeds Memory Access Times", Electronics, January 12, 1984, pp. 164-169.

[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.

[Gust82] R. N. Gustafson and F. J. Sparacio, "IBM 3081 Processor Unit: Design Considerations and Design Process", IBM J. Res. and Devel., 26, 1, January, 1982, pp. 12-21.

[Hrus82] J. Hrustich and W. R. Sitler, "Dual-Stream Processor Cache Search Synchronization", IBM Tech. Disc. Bull., 25, 7A, December, 1982, pp. 3559-3561.

[IBM85] IBM Corp., "Shared Instruction and/or Data Caches in a Multiprocessing System", IBM Tech. Disc. Bull., 27, 12, May, 1985, pp. 6845-6846.

[Jone76] J. D. Jones, D. M. Junod, R. L. Partridge and B. L. Shawley, "Updating Cache Data Array's with Data Stored by Other CPU's", IBM Tech. Disc. Bull., 19, 2, July, 1976, pp. 594-596.

[Jone77] J. D. Jones and D. M. Junod, "Cache Address Directory Invalidation Scheme for Multiprocessing System", IBM Tech. Disc. Bull., 20, 1, June, 1977, pp. 295-296.

[Katz85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon, "Implementing a Cache Consistency Protocol", Proc. 12'th Ann. Int. Symp. on Comp. Arch., June, 1985, Boston, Mass, pp. 276-283.

[Knig84] J. W. Knight and L. Liu, "Early Store-Through of XI-Sensitive Data", IBM Tech. Disc. Bull., 27, 2, July, 1984, pp. 1073-1074.

[Kryg84] M. A. Krygowski, "Probabilistic Updating for Store-In Cache Cross Interrogation", IBM Tech. Disc. Bull., 26, 10B, March, 1984, pp. 5504-5505.

[Maza77] Guy Mazare, "A Few Examples of How to Use a Symmetrical Multi-Micro-Processor", Proc. 4'th Ann. Symp. on Computer Architecture, March, 1977, pp. 57-62.

[Nest85] Elliot Nestle and Armond Inselberg, "The Synapse N+1 System: Architectural Characteristics and Performance Data of a Tightly Coupled Multiprocessor System", Proc. 12'th Ann. Int. Symp. on Computer Architecture, June 17-19, 1985, Boston, Mass., pp. 233-239.

[Papa84] Mark Papamarcos and Janak Patel, "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories", Proc. 11'th Ann. Int. Symp. on Comp. Arch., June, 1984, Ann Arbor, Michigan, pp. 348-354.

[Puza83] T. R. Puzak, R. N. Rechtschaffen and K. So, "Managing Targets of Multiprocessor Cross Interrogates", IBM Tech. Disc. Bull., 25, 12, May, 1983, p. 6462.

[Rudo84] Larry Rudolph and Zary Segall, "Dynamic Decentralized Cache Schemes for MIMD Parallel Architectures", Proc. 11'th Ann. Int. Symp. on Comp. Arch., June, 1984, Ann Arbor, Michigan, pp. 340-347.

[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, "Problems, Directions and Issues in Memory Hierarchies", Proc. 18'th Annual Hawaii International Conference on System Sciences, January 2-4, 1985, Honolulu, Hawaii, pp. 468-476. Also available as UC Berkeley CS Report UCB/CSD84/220.

[Smit85b] 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.

[Smit85c] Alan Jay Smith, "Line (Block) Size Selection in CPU Cache Memories", June, 1985. Available as UC Berkeley CS Report UCB/CSD85/239.

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

[Yeh81] Chi-Chung Yeh, "Shared Cache Organization for Multiple-Stream Computer Systems", Report R-904, January, 1981, University of Illinois Coordinated Science Laboratory.

[Yeh83] Phil Yeh, Janak Patel and Edward Davidson, "Performance of Shared Cache for Parallel-Pipelined Computer Systems", Proc. 10'th Ann. Int. Symp. on Comp. Arch., Stockholm, Sweden, June, 1983, pp. 117-123.

[Yen85] Wei Yen, David Yen, and King-Sun Fu, "Data Coherence Problem in Multicache System", IEEETC, C-34, 1, January, 1985, pp. 56-65.