Cache Memory Optimization to Reduce Processor/Memory Traffic

by

James R. Goodman

Computer Sciences Technical Report #580

February 1985

Cache Memory Optimization to Reduce Processor/Memory Traffic

James R. Goodman

Department of Computer Sciences
University of Wisconsin-Madison
Madison, WI 53706

January 29, 1985

ABSTRACT--The importance of reducing processor-memory bandwidth is recognized in two distinct situations: single board computer systems and microprocessors of the future. Cache memory is investigated as a way to reduce the memory-processor traffic. We show that traditional caches which depend heavily on spatial locality (look-ahead) for their performance are inappropriate in these environments because they generate large bursts of bus traffic. A cache exploiting primarily temporal locality (look-behind) is then proposed and demonstrated to be effective in an environment where process switches are infrequent. We argue that such an environment is possible if the traffic to backing store is small enough that many processors can share a common memory and if the cache data consistency problem is solved. We demonstrate that such a cache can greatly reduce traffic to memory, and introduce an elegant solution to the cache coherency problem.

Submitted to
ACM Transactions on Computer Systems
January 29, 1985

1. Introduction

It is widely recognized that advances in semiconductor technology have had a powerful impact on computing. Its influence, however, has not been uniform across the computing spectrum. In particular, microcomputer-based systems have seen enormous improvements in cost-effectiveness, producing seemingly endless miracles for consumer products. The impact on larger, more powerful computers has been much less dramatic. For computers with a CPU implemented on more than one chip, by far the most important advance has been a reduction in memory cost. While microprocessors also have access to equally cheap memory, it is relatively more expensive in these systems because the CPU cost is so low. This has resulted in different attitudes toward memory in microprocessor systems as opposed to minicomputers and mainframes. Indeed, a major distinction between minicomputers and microprocessors now is the amount of memory they access, both virtual and real.

What is the future direction of computer memory systems? We would like to suggest that there is a fundamental distinction between those for single-chip processors and those for CPUs implemented with many integrated circuits. The limitations imposed by interchip communications are so substantial compared to internal communications that this distinction must be one of the most important considerations in the design of a new computer architecture. For single-chip processors, memory will always be more expensive relative to computing power than for multi-chip CPUs.

1.1. A Very-High-Performance Single-Chip Processor

It will soon be possible to fabricate an extremely high-performance CPU on a single chip, roughly the size of today's mass-produced microprocessors. However, devoting the entire chip to the CPU is not a good idea. The communications necessary to utilize effectively such a powerful processor are substantially greater than that feasible in a 64-pin, or even 100-pin, package. Though there is substantial interest in packaging techniques that greatly expand the number of pins, this advance brings with it major problems. Among them are

(1) a decrease in reliability, which is closely related to the number of pins, and

(2) an increase in power consumption. An increasing proportion of the total power of a chip is consumed driving signals off-chip.

For these reasons, we believe that the total communications bandwidth onto a VLSI chip will become an increasingly severe limitation.

Extrapolating historical trends to predict future component densities, in a few years we might expect years to purchase a single-chip processor containing at least ten times as many transistors as in, say, the MC68020. For the empirical rule known as Grosch's law [Grosch53], $P = k\ C^{\ g}$, where P is some measure of performance, C is the cost, and k and g are constants, Knight[Knight66] concluded that g is at least 2, and Solomon[Solomon66] has suggested that $g \approx 1.47$. For the IBM System/370 family, Siewiorek determined that $g \approx 1.6$ [Siewiorek82]. While Grosch's law breaks down in the comparison of processors using different technology or architectures, it is realistic over a limited range for predicting improvements within a single technology. Siewiorek in fact suggests that it holds "by definition."

Assuming $g = 1.5$, using processor-memory bandwidth as a measure of performance, and number of transistors (or gates) as a measure of cost, Grosch's law predicts that a processor containing 10 times as many transistors as a current microprocessor would require 30 times the memory bandwidth.[1] The Motorola MC68000, running at 12.5 MHz, accesses data from memory at a maximum rate of 6.25 million bytes per second, using more than half its pins to achieve this rate. Although packaging technology is rapidly increasing the pins available to a chip, it is unlikely that the increase will be 30-fold (the 68000 has 64 pins). We would suggest a factor of two is realistic. Although some techniques are clearly possible to increase the transfer rate into and out of the 68000, supplying such a processor with data as fast as needed is a severe constraint. One of the designers of

[1] This is a conservative estimate, in fact, because it ignores predictable decreases in gate delays.

the 68000, has stated that all modern microprocessors - the 68000 included - are already bus-limited [Tredennick82].

1.2. On-chip Memory

One alternative for increased performance without proportionately increasing processor-memory bandwidth is to introduce memory on the same chip with the CPU. Several recently announced, high-performance microprocessors, *(e.g.*, the MC68020 and Z80000), include an on-chip cache. While devoting the entire chip to the CPU could result in a more powerful processor, introducing on-chip memory offers a reduction in memory access time due to the inherently smaller delays as compared to inter-chip data transfers. If most accesses are on-chip, the slower processor can outperform the more powerful one because of the reduced memory latency.

Ideally, the chip should contain as much memory as the processor "needs" for main storage. Conventional wisdom today says that a processor of the speed of current microprocessors needs at least 1/4 megabytes of memory [Lindsay81]. This is certainly more than is feasible on-chip, though a high performance processor could make good use of several megabytes. Clearly in the foreseeable future, all the primary memory for the processor cannot be placed on the same chip with a powerful CPU. What is needed is the top element of a memory hierarchy.

The remainder of the paper is organized as follows. Section 2 discusses the issue of bus traffic between the cache and main memory. Section 3 discusses the architecture of single-board computers, and limitations imposed by the backplane bus. Section 4 discusses the issue of data consistency in the presence of multiple cache memories, and describes a technique appropriate for single-chip or single-board computers for assuring consistency. Section 5 presents simulation results for a variety of cache designs. Section 6 contains our conclusions.

2. Cache Memory

Cache memory is effective because it exploits the non-random nature of memory reference patterns. This is usually characterized as two types of locality, *spatial* and *temporal*. Spatial locality reflects the fact that memory locations with small addresses relative to a location just accessed are much more likely than average to be accessed in the near future. Temporal locality reflects the fact that references to a given location tend to be clustered in time.

The use of cache memory, however, has often aggravated the bandwidth problem rather than reduce it. Spatial locality is generally exploited by fetching a large block of data whenever a miss occurs. As a result, often a larger burst bandwidth is required from main storage to the cache than would be necessary without a cache. For example, the cache on the IBM System/370 model 168, receives data from main memory at a burst rate of 100 megabytes per second [IBM76], supplying data to the CPU at less than 1/3 that rate. This is done in order to exploit the spatial locality in memory references, the data being transferred from backing store into the cache in large blocks, or *lines*, but resulting in requirements for very high bandwidth bursts of data. We measured the average bandwidth on an IBM System/370 model 155, and concluded that the *average* backing-store-to-cache traffic is less than the cache-to-CPU traffic.

The cost of providing a high-bandwidth path from main storage to a cache (and the high-bandwidth memory itself), while reasonable for main-frame computers, is relatively more expensive for mini-computers. Lowering the bandwidth from backing store to the cache can be accomplished in one of two ways:

(1) bring in small blocks of data from backing store to the cache, or

(2) accept long delays while a block is being brought in, independent of (and in addition to) the access time of the backing store.

While it is possible to bring in the word requested initially (read through), thus reducing the wait on a given reference, the low bandwidth memory interface will remain busy long after the initial transfer is completed, resulting in long delays if a second backing storage operation is required. This also

creates complicated bookkeeping operations if a cache is repeatedly accessed while a line is partially filled.

The designers of the PDP-11 models 60 and 70 chose small line sizes to keep main memory traffic low [Bell78]. This choice reduces cache effectiveness, particularly at the time of a context switch, when spatial locality can best be exploited. It apparently was made to reduce cost, but even so, a new cache/main memory bus was introduced for systems including a cache.

In the single-chip processor, increasing the off-chip memory bandwidth is extremely expensive, and high bandwidth is simply unavailable. Thus spatial locality is difficult to exploit and we must rely on temporal locality for the cache effectiveness. We have explored cache memory which exploits primarily or exclusively temporal locality, *i.e.*, the blocks fetched are near the minimum size possible.

3. The Single-Board Computer

It happens that the single-board-computer business -- already a major market -- contains the same basic constraint: processor/memory bandwidth. Though several standard buses exist, by far the most popular is IEEE Standard 796-1983, which is essentially the Multibus [2]

A single board computer typically contains a microprocessor and a small number of memory chips. If needed, access to additional random access memory is through a backplane bus, which is designed for generality and simplicity, not for high performance. Multibus, in particular, was defined in the early 70s to offer an inexpensive means of communication among a variety of subsystems. Nearly a thousand different Multibus cards are available from hundreds of vendors.

While the market has rapidly developed for products using this bus, its applications are limited by the severe constraint imposed by the bandwidth of Multibus. Clearly the bus bandwidth can be increased by increasing the number of pins, and by modifying the protocol. This has in fact been done with Multibus II [Intel]. The broad popularity of Multibus and the availability of components to implement its protocol suggest, however, that it is likely to survive many years in its present form. Thus a large market exists for a computer-on-a-card which, much as if it were all on a single chip, has severe limitations on its communications with the rest of the system.

Multibus systems have generally dealt with the problem of limited bus bandwidth by removing most of the processor-memory accesses from the bus. Each processor card has its own local memory, which may be addressable to others through the Multibus. While this approach might be considered a user-managed cache, we believe that the allocation of memory - local or remote - should be handled by the system, freeing the programmer of this task. In typical Multibus applications, much effort is expended guaranteeing that the program running is primarily resident on the same card with the CPU. This approach is viable for a static partitioning of tasks. Systems trying to allocate processors dynamically have generally found it necessary to include a higher-performance bus [Isaack82, Frank84].

In many environments, a simple dynamic hardware allocation scheme can efficiently determine what memory locations are being accessed frequently and should therefore be kept in local memory - better than the programmer who often has little insight into the dynamic characteristics of his program. There are environments where the programmer is intimately familiar with the behavior of his program and can generate code to take advantage of it. In this environment the time spent running a program is often much more substantial than the time developing the program. This explains, for example, why an invisible[3] cache is not appropriate on the CRAY-1. Freeing the programmer from concern about memory allocation is essential where programmer productivity is critical, however.

We suggest as an alternative a single-board computer containing, (possibly along with other things) a CPU and no local memory except a cache, with backing store provided through Multibus.

---

[2] Multibus is a trademark of Intel Corporation

[3] Some would argue that the large number of registers in the CRAY-1 architecture constitute a user-visible cache.

While several commercial products exist which have a microprocessor and cache memory on a single board, all the systems of which the author is aware involve the use of a much higher speed backplane bus. Particularly important is the ability for multiple processors -- including some not containing cache -- to work in parallel. Thus the well-known consistency problem resulting from memory redundancy must be solved. We present a solution in section 4.

We believe that a system which could reasonably support 5 to 10 such processors would be a significant advance. This can't be compared directly against current systems because a single processor overloads the Multibus. Thus local memories must be heavily exploited if performance is important.

Earlier cache studies [Kaplan73, Bell74, Rao78, Patel82] have used the cache hit ratio or something closely related to measure performance. The important criterion here is to maximize use of the bus, not the hit ratio, or even necessarily to optimize processor performance. We optimize system performance by optimizing bus utilization, achieving higher performance by minimizing individual processors' bus requirements, and thereby supporting more processors. We allow individual processors to remain idle periodically rather than create bus traffic prefetching data which they might not use.

We have identified two distinct computer classes -- single-chip, high-performance processors of the future and single-board computers using standard backplane buses -- each of which has as a major constraint, not the memory access time, but rather the *rate* at which it supplies data to the processor.

4. Cache Coherency

It is well-known that multiple caches present serious problems because of the redundancy of storage of a single logical memory location [Tang76, Censier78, Rao78]. The most common method among commercial products for dealing with this, the stale data problem, is to create a special, high-speed bus on which addresses are sent whenever a write operation is performed by any processor. This solution has weaknesses [Censier78] which have generally limited commercial implementations to two caches. In the single-chip processor or single-board computer environments, it has the added weakness that it involves additional communications bandwidth off-chip.

An alternative approach, implemented in C.mmp [Hoogendoorn77] the Honeywell Series 66, and Elxsi 6400, is to require the operating system to recognize when inconsistencies might occur and take steps to prevent them. This solution is unappealing because the cache is normally regarded as an architecture-independent feature, invisible to the software.

A third approach, variations of which have been proposed by Censier and Feautrier [Censier78], Tang [Tang76], Widdoes [Widdoes79], and Yen and Fu [Yen82], is to use some form of tagged main memory, keeping track of individual lines to prevent inconsistency. An individual line is temporarily designated as *private* for a particular processor so that it may modify it repeatedly without reference to main memory. The tag must be set whenever such a critical section is entered and reset whenever the critical section is left, *i.e.*, the modified word is written back to main storage. This approach requires substantial hardware, and appears infeasible for a large number of caches, since an operation in a central place is required at the entry or exit of any critical section, though a nice simplification of this approach has been proposed which reduces the hardware requirements [Archibald84].

Another approach allows the critical section information to be distributed among the caches, where it already resides. The Synapse N+1 architecture [Frank84] uses a tag bit in main memory to keep track of ownership of 16-byte lines. By the use two distinct read requests, private and public, the ownership is transferred to individual caches. A related scheme [Amdahl82] which uses a special bus to convey the notice of entry or exit from a critical section, has been implemented in a commercial product, but has not been published to our knowledge.

Our approach [Goodman83, Ravishankar83] has much in common with these approaches, but uses the normal read and write operations, with no tag bits in main memory, to accomplish the synchronization. It is called ***write-once***.

4.1. Write Strategy

Two strategies are generally recognized for handling write operations in a cache. One, often referred to as *write-through* or *store-through*, stipulates that all writes are immediately recorded in main memory. The alternative, variously called *write-back*, *store-back*, or *copy-back*, allows data to be written temporarily only to the cache, memory being updated before the cache line is purged. Write-through has the advantages that it is simpler to implement and that main memory is always current. The latter is an important point because it means that only error *detection* is necessary within the cache. Write-back generally has the advantage of fewer main-storage operations.

While the choice between write-through and write-back has no bearing on the read hit ratio, it has a major impact on bus traffic, particularly as the hit ratio approaches 100%. In the limit, when the hit ratio is 100%, write-back results in no bus traffic at all, while write-through requires at least one bus cycle for each write operation. Norton [Norton82] concluded that using write-back instead of write-through for a hypothetical processor typically would reduce the bus traffic by more than 50% and if the processes ran to completion bus traffic would be decreased by a factor of 8. Pier [Pier83] indicates that using write-back reduces traffic between the cache and main storage of the Dorado by a factor of about seven for write operations. For typical read-to-write and hit ratios and when task switching is infrequent, our simulations have confirmed that write-back generates substantially less bus traffic than write-through.

But write-back has more severe coherency problems than write-through, since even main memory does not always contain the current version of a particular memory location.

4.2. A New Write Strategy: Write-Once

We describe a write strategy which solves the stale data problem and produces minimal bus traffic. While the scheme was developed to assure coherency among multiple caches and a main memory, a surprising result is that it consistently generates bus traffic as low as the better of the two traditional strategies, and in many cases is superior to both. The replacement technique requires the following structure. Associated with each line in the cache are two bits defining one of four states for the associated data:

Invalid — There is no data in the line.

Valid — There is data in the line which has been read from backing store and has not been modified.

Reserved — The data in the line has been locally modified exactly once since it was brought into the cache and the change has been transmitted to backing store.

Dirty — The data in the line has been locally modified more than once since it was brought into the cache and the latest change has not been transmitted to backing store.

As with most schemes that assure coherence, the tag memory must be queried periodically because of activity other than from the attached CPU. Write-once requires this request to be serviced quickly in order to assure consistency without delaying normal bus accesses. This can most easily be achieved by creating two (identical) copies of the tag memory. Censier [Censier78] claims that duplication is "the usual way out" for resolving collisions between cache invalidation requests and normal cache references, though we are unaware of any commercial products employing this method. It is a significant cost, particularly for a single-chip processor, but is necessary. In section 5.5 we show that the size of the tag memory can actually be kept quite small despite the large number of tags implied by a small line size. For the single-board computer, the cost could be as low as a single replicated chip, though such a chip is not commercially available. (The TMS2150 [TI84] chip is similar to what is needed but does not support write-back).

The two copies of the tag memory always contain exactly the same data, because they are always written simultaneously. While one unit is used in the conventional way to support accesses by the CPU, (*i.e.*, any state but *invalid* signifies a valid address), a second monitors all accesses to memory via the Multibus. For each such operation, it checks for an address match in the local cache. If a match is found on a write operation, it notifies the cache controller, and the entry in both tag memories is marked *invalid*. This should be done as soon as possible, and must be done before any write requests from the CPU are satisfied. Finding a *modified* tag at this location indicates an error.

If a match is found on a read operation, nothing is done unless the line has been modified, *i.e.*, its state is *reserved* or *dirty*. If it is *dirty*, the local system inhibits the backing store from supplying the data. It then supplies the data itself.[4] On the same bus access or immediately following it, the data must be written to backing store. In addition, for either *reserved* or *dirty* data, the state is changed to *valid*.

This scheme achieves coherency in the following way. Write allocation is employed in the cache. This means that any modification to data can only be performed if it is in the cache, and results in a fetch on a cache miss (the data may come from another cache). For unmodified (*valid*) data in the cache, write-through is employed. The resulting bus operation achieves an additional goal: purging the copies from all other caches. The cache writing through the bus is now guaranteed the only copy except for backing store. This is the meaning of the state *reserved*: "written once." If it is purged at this point, no write is necessary to backing store. Since main storage contains the correct value, it may be treated the same as *valid* data (and marked *valid*) when accessed through the bus. If another write occurs locally, the *reserved* line is marked *dirty*. Now write-back is employed and no bus activity required. The local copy is the sole correct copy, however, and when purged (or downgraded to *valid*), must be written to backing store.

Write-once has the desirable feature that units accessing backing store need not have a cache, and need not know whether others do. A cache is responsible for maintaining consistency exactly for those cases where it might create a violation, *i.e.*, those lines that it modifies. Thus it is possible to mix in an arbitrary way systems which employ a cache and those which do not; the latter would probably be I/O devices. Considerable care must be exercised, however, when a write operation over the bus modifies less than an entire line.

5. Simulation

To evaluate the effectiveness of cache memory in reducing processor/memory traffic we evaluated the memory traffic for a variety of conditions using trace-driven simulation. We generated and used six different traces for a VAX-11[5] architecture running UNIX[6] version 4.2bsd:

NROFF	The program *nroff* interpreting the Berkeley macro package *-me*.
CACHE	The trace-driven cache simulator program.
COMPACT	A program using an on-line algorithm which compresses files using an adaptive Huffman code.
AS	The standard UNIX (VAX-11) assembler.
GREP	The UNIX string matching program *grep*, searching through the dictionary for a nonsense string.

---

[4] There is a mechanism in Multibus which allows this capability. Unfortunately, it is rarely used, poorly defined, and requires that local caches respond very rapidly. Other standard buses have cleaner mechanisms by which this end can be accomplished, usually involving a "retry" procedure.

[5] VAX is a trademark of the Digital Equipment Corporation.

[6] UNIX and NROFF are trademarks of Bell Laboratories.

SORT The standard UNIX sorting program sorting the first 1000 entries in the dictionary.

The VAX is an example of a modern microcoded instruction set and, therefore is a reasonable example of one kind of processor likely to appear in a single-chip CPU in the future. The trace program actually generates virtual addresses, but all of the programs we ran are small enough to fit easily into a million bytes of main memory. Since we are tracing only a single process, we conclude that there is no significant difference between virtual and real addresses.

The traces were collected on a VAX-11 by a program using the UNIX *ptrace* system call, which sets the VAX trace bit to trap after each instruction. The instruction is then interpreted and memory references are recorded. Thus the trace represents a single process executing without interruption. This is unrealistic in a time-sharing environment, such as the one used to collect and evaluate the data, where frequent interruptions occur for page faults, task-switching, and terminal handling. We argue here, however, that in the single processor environment of the future, single processes may actually run for long periods without interruption, so our trace is perhaps not unrealistic for the environment under investigation. It *did* result in substantially more optimistic predictions than direct measurements have indicated [Clark83].

An additional effect, however, of the method used for tracing is that any system call appears in the trace as a single instruction. When a system call requires changing the mode to kernel, *(i.e.*, a *CHMK* instruction is executed,) the tracing is turned off until the occurrence of the corresponding return *(REI)*, which restores the user mode. A large portion of all cache misses occur either on a task switch or in the operating system [Smith82]. Treating the CHMK instruction like any other may result in serious distortion of the actual memory reference pattern in two ways:

(1) The system code executed may exhibit different memory reference behavior than the programs being traced. Being unable to trace programs running in kernel mode, we have so far been unable to determine this.

(2) Upon the return from the system call, the cache is likely to have been radically changed, possibly completely flushed. This is particularly significant for very small caches.

In this study we have made no attempt to evaluate the behavior of kernel programs, but we did bracket the effect of the internally generated system call by evaluating two extreme cases: (1) treating CHMK like any other instruction and (2) flushing the cache completely on every CHMK instruction. This had relatively little effect (less than 1% difference) on small caches because of the infrequency of the CHMK instruction (0.013% of all instructions executed), but somewhat more effect on large caches because of the long time to refill them. The miss ratio was approximately 30% lower for cold start, 60% lower for warm start for the former case. For small block sizes the first assumption resulted in numbers 15-25% lower for cold start, 20-50% lower for warm start. For large block sizes the differences were less than 1%. In the tables we report only the times for the more pessimistic assumption, *viz.*, that the cache is flushed every time a system call occurs. We point out that the effect of virtual memory may result in substantially more pessimistic results than our cold start measurements indicate if additional interruptions are necessary because of page faults.

Cache performance is greatly affected by cache parameters, particularly total size and line size. In addition, performance varies greatly depending on the program running. For each of the above traces, a wide and unpredictable variation occurred as we varied a single parameter. Thus plotting parameters for the individual traces was often not enlightening. Averaging over the traces in each category gave much more revealing results, providing data that suggested a continuous function for many of the variables studied. Thus all our results are reported as the mean of the programs, each running alone.

5.1. Assumptions and Default Cache Parameters

Clark has pointed out serious limitations for the simulation approach to cache evaluation [Clark83]. A major advantage over measurement, however, is the relative ease of varying numerous

cache parameters, simulating a wide variety of cache designs. The actual sequence of memory references can only be approximated by simulation, however, since the interlacing of instruction fetching and data fetching for most computers is very hard to predict. It is also different for different implementations of the VAX architecture. For the VAX-11/780, the instruction buffer fetches data by a simple algorithm [Clark83] but one for which the number of instruction fetches varies depending on many things, including the cache behavior. Thus it is impossible to predict even the number of instruction fetches independently of the cache definition. Again, while it can be argued that this affects the absolute miss ratio, the relative performance of different cache designs should compare realistically. We made the following assumptions regarding the implementation:

(1) The individual bytes of the instruction are fetched as they are interpreted by the microcode. However, a four-byte, longword-aligned, instruction buffer is assumed, so that the cache always supplies four bytes at a time, and the same longword is never fetched twice consecutively.

(2) All data locations fetched during the execution of a single instruction are fetched exactly once. All data locations stored within a single instruction are written exactly once. Within a single instruction, we assumed that the microcode was smart enough to optimize the reads and writes within a single longword.

5.1.1. Effect of Writes on Miss Ratio

The miss ratio calculation includes writes. Excluding them has no significant effect.

5.1.2. Write Allocation

Write allocation, also known as *fetch on write*, means that a line is allocated in the cache on a write miss as well as on a read miss. While it seems natural for write-back, it typically is not used with write-through. It is essential for write-once to assure coherency. Our early simulations showed that it was highly desirable for write-back and write-once, and superior even for write-through with small lines. This was true using both the measures of miss ratio and bus traffic. In all results presented, write allocation was employed.

5.1.3. Associativity

We ran a number of simulations varying the associativity all the way from direct mapped to fully associative. While this is clearly an important parameter, we found no unexpected correlation between this and the other parameters studied. Again, the associativity may affect the absolute miss ratios, but not the relative ones. For all results reported, the cache was assumed to be 4-way set-associative

5.1.4. Replacement Algorithm

Replacement strategy has been the subject of other studies using the same simulator and traces [Smith83, Smith85]. In order to limit its significance, which seems to be orthogonal to the issues raised here, we have assumed true LRU replacement among the four lines in each set

5.1.5. Bus Width

The width of the data paths between units is an important parameter being related to, but not the same as, memory bandwidth. The bus traffic is given as a percent of the number of accesses that would be required if no cache were present, and assumes both the CPU/cache bus and the cache/main memory bus are 4 bytes wide. An 8-byte transfer therefore is counted as two cycles.

Most microprocessors today provide bi-directional pins for moving the data on and off the chip, and some also multiplex addresses on these pins as well. Bandwidth is therefore strongly affected by access time, and fetching two words from memory typically takes twice as long as fetching one (barring an external cache, of course). While higher bandwidth can result from a pair of unidirectional

buses -- one supplying addresses and write data, the other receiving read data -- backplane buses such as Multibus also generally tie up the bus for the duration of a fetch operation. We therefore assume a memory access delay proportional to the block size. In section 5.5 we address the question of a block transfer across the bus and its implication on the appropriate block size.

5.1.6. Default Parameters

In the tables that follow, unless otherwise stated, the cache size was 4K bytes. The organization was four-way set-associative with LRU replacement. The write strategy was write-once (with write-allocation) and the line size was 4 bytes. All simulations are the average of the same 600,000 memory references. 100,000 from each of the six traces. Boldface entries generally indicate the most desirable choices.

5.2. Cache Size

Cache size is almost certainly the single most important parameter affecting cache performance. Our simulations confirmed this well-known result. Both the bus traffic and the miss ratio, which show high correlation when this parameter is varied, show steady improvement as the cache size is increased. The miss ratio declined 20-30% as the cache was increased from 4K bytes to 8K. Under steady-state conditions, it even declined when the cache was increased from 32K bytes to 64K, though it is so low already by that point that system flushes (if included) are the dominant cause of misses.

Though early implementations of on-chip caches are quite small, the size of cache memory will steadily grow over time. We believe that the improved performance -- both miss ratio and bus traffic -- will soon warrant a cache of substantial size. We arbitrarily chose the values of 4K and 16K bytes for the simulation studies because we believe such sizes represent a realistic range for single-board and single-chip implementations.

5.3. Cold Start vs. Warm Start

Based on the work of Easton and Fagin [Easton78] we define the *cold start period* as the number of memory references from an initially empty cache until *CAP* misses have occurred, where *CAP* is the capacity of the cache (in lines). (This is the point at which data from a fully associative cache with LRU replacement is first purged). This initial burst of misses is amortized over all

Table 1: MISS RATIO vs. CACHE SIZE

Cache Size (Bytes)	4-Byte Lines		8-Byte Lines		16-Byte Lines	
	Cold Start	Warm Start	Cold Start	Warm Start	Cold Start	Warm Start
64	81.7	69.7	63.2	51.3	**59.6**	**43.3**
128	62.1	50.2	45.6	36.0	**43.3**	**31.5**
256	46.7	36.3	34.4	25.4	**29.7**	**20.5**
512	34.0	24.0	23.8	16.9	**17.7**	**12.5**
1024	18.3	11.4	14.7	8.86	**12.3**	**7.73**
2048	9.36	5.89	7.19	4.40	**6.43**	**3.81**
4096	5.44	4.39	3.83	2.91	**2.92**	**2.04**
8192	4.33	3.70	2.67	2.30	**1.76**	**1.49**
16384	3.66	3.33	2.37	2.01	**1.47**	**1.36**
32768	3.52	3.21	2.05	1.88	**1.24**	**1.14**
65536	3.52	3.21	2.05	1.88	**1.23**	**1.13**

Table 2: BUS TRAFFIC RATIO vs. CACHE SIZE						
Cache Size (Bytes)	4-Byte Lines		8-Byte Lines		16-Byte Lines	
	Cold Start	Warm Start	Cold Start	Warm Start	Cold Start	Warm Start
64	**86.3**	**75.1**	131	114	243	186
128	**65.5**	**55.3**	94.8	81.1	178	139
256	**50.0**	**41.3**	71.9	58.7	123	93.0
512	**36.9**	**27.3**	50.0	38.9	73.6	57.4
1024	**20.0**	**13.2**	31.3	20.5	51.3	35.2
2048	**10.4**	**7.03**	15.3	10.3	27.0	17.5
4096	**6.17**	**5.17**	8.25	6.80	12.3	9.24
8192	**4.91**	**4.26**	5.76	5.14	7.44	6.54
16384	**4.22**	**3.84**	5.11	4.36	6.17	5.73
32768	**4.07**	**3.71**	4.43	4.05	5.18	4.75
65536	**4.06**	**3.71**	4.43	4.04	5.13	4.70

accesses, so the longer the trace analyzed, the lower the miss ratio obtained. In addition to the initiation of a program and occasional switches of environments, a cold start generally occurs whenever there is a task switch. (At least a partial cache flush occurs on a system call, as discussed earlier). Thus an important consideration in traditional cache evaluation is the frequency and distribution of task switches. We have argued that task switching must be very infrequent to minimize bus traffic.

To assess the significance of filling the cache we attempted to separate the steady-state performance from transient behavior, encountered whenever the cache contains little useful data. To do this, we collected statistics for each of the six traces under two different conditions.

(1) *Warm Start* We simulated 200,000 memory references in the cache, but collected statistics starting after 100,000 references, at which point the cache has long since reached steady-state for all situations studied.

(2) *Cold Start* We skipped 100,000 memory references, then started simulation with an empty cache. In addition, we flushed the cache whenever the accumulated number of misses reached the total number of lines in the cache, *i.e.*, at the end of each measure cold start period.

The results are shown in tables 1-4. In tables 1 and 2, the miss and bus traffic ratios respectively are shown for a variety of cache sizes, with line sizes of 4, 8 and 16 bytes. Tables 3 and 4a and 4b show the miss ratio and bus traffic respectively for various line sizes.

5.4. Effect of Write Strategy on Bus Traffic

Although write-through normally generates less bus traffic than write-back, the latter can be worse if the hit ratio is low and the line size is large. Under write-back, when a dirty line is purged, the entire line must be written out. With write-through, only that portion which was modified must be written. We found that write-back is decisively superior to write-through except (1) when cache lines are very large, or (2) when the cache size is very small.

Write-once results in bus traffic roughly equal to the lower of the two. For a number of cases it actually performs better on the average than either write-through or write-back. This is a surprising result, since write-once was developed to assure coherency, not to minimize bus traffic. Unfortunately, this occurs when relatively large lines are used, and write-back is slightly superior for small blocks. See tables 4a and 4b.

Table 3: MISS RATIO vs. LINE SIZE				
Line Size (Bytes)	4K Bytes		16K Bytes	
	Cold Start	Warm Start	Cold Start	Warm Start
4	5.44	4.39	3.66	3.33
8	3.83	2.91	2.37	2.01
16	2.92	2.04	1.47	1.36
32	**2.71**	**1.68**	0.892	0.795
64	2.95	1.76	0.627	0.528
128	3.70	1.92	**0.542**	**0.398**
256	4.59	2.53	0.710	0.432

Table 4a: BUS TRAFFIC RATIO vs. LINE SIZE Cache size is 4K Bytes.						
Line Size (Bytes)	Write Back		Write Once		Write Through	
	Cold Start	Warm Start	Cold Start	Warm Start	Cold Start	Warm Start
4	**4.94**	**4.42**	6.17	5.17	17.9	17.0
8	**7.49**	**6.27**	8.25	6.80	20.4	18.7
16	**11.8**	**9.04**	12.3	9.24	24.6	21.1
32	**22.4**	15.3	22.6	**15.1**	34.6	26.5
64	50.2	33.6	**49.2**	**31.8**	60.0	41.0
128	126	76.8	**122**	**70.6**	131	74.5
256	312	210	**302**	189	306	**175**

Table 4b: BUS TRAFFIC RATIO vs. LINE SIZE Cache size is 16K-Bytes.						
Line Size (Bytes)	Write Back		Write Once		Write Through	
	Cold Start	Warm Start	Cold Start	Warm Start	Cold Start	Warm Start
4	**3.18**	**2.88**	4.22	3.84	16.3	16.0
8	**4.53**	**3.83**	5.11	4.36	17.6	16.9
16	**5.84**	**5.44**	6.17	5.73	18.9	18.5
32	**7.16**	**6.47**	7.35	6.62	20.2	19.4
64	10.5	9.10	**10.5**	**9.02**	23.1	21.5
128	18.7	14.5	**18.6**	**14.3**	30.4	25.8
256	50.6	34.6	**50.0**	**33.4**	58.4	40.7

5.5. Line Size

Our cache design incorporates small lines, depending heavily on temporal locality. The results shown in tables 1 and 2 assumed a line size of four to 16 bytes. Easton and Fagin [Easton78] argue that for a sufficiently large cache, line size has no effect on the warm start miss ratio, but strongly affects the cold start miss ratio. They argue that if the cache can hold all the data required, misses occur only during the cold start period. Our simulations found that line size affects miss ratio under warm or cold start conditions. In fact, the miss ratio is *more sensitive* to line size variations under

warm start conditions. Keeping the total size constant (4 K bytes), as line size is increased the miss ratio generally declines up to a point -- 32 bytes for a 4K-byte cache, 128 bytes for a 16K-byte cache -- then increases for either warm or cold starts. Under cold start conditions for a 4K-byte cache, the cold start miss ratio declines by 50% as line size is increased from 4 to 32 bytes. Under steady-state conditions, it declines 62%. For a 16K-byte cache the effect is even more dramatic: for cold start it declines 75% going from 4- to 32-byte lines, and 85% for 128-byte lines. For warm start the numbers are 76% and 88% respectively. Thus we conclude that 32-byte lines result in the minimum miss ratio under either warm or cold start conditions for a 4K-byte cache. For a 16K-byte cache the minimum miss ratio is achieved with 128-byte lines. See Table 3.

The bus traffic to main memory is affected differently by the line size. When the line size is doubled, the miss ratio must be halved to maintain constant bus traffic. It is nearly impossible for the miss ratio to be cut by more than 50%, and our simulations show that it rarely approaches that. Consequently, the bus traffic always increases as the line size is increased. As the effect on the miss ratio of increasing line size declines, the bus traffic grows more rapidly, so that the bus traffic becomes very large even before the miss ratio starts to increase. For 32-byte lines the bus traffic is two to four times that for 4-byte lines under all conditions except when write-through is used (both numbers are large, being dominated by writes). Clearly smaller lines are needed when bus traffic is a potential bottleneck.

The bus traffic under transient conditions is higher than for steady-state conditions, though not by much for small line sizes. This suggests that even if task switching occurs frequently, a cache with small lines can significantly reduce bus traffic to/from memory.

Our simulations show that reducing the line size to a single transfer across the bus decreases the hit ratio, decreasing bus traffic roughly in proportion. Increasing the transfer line size from one bus cycle (four bytes) to two (eight bytes) decreases the miss ratio by 30 to 40%, increasing the bus traffic by a similar amount. These results compare favorably with those reported by Strecker for the PDP-11/70 [Bell78].

For the results reported above we have made the assumption that access time is related linearly to line size. In many cases this is not true. It is essentially true for the Multibus, since only two bytes can be fetched at a time, though arbitration is overlapped with bus operations. For a single-chip implementation, higher bandwidth can be achieved by providing the capability for efficient multiple transfers over a set of wires into the processor. We evaluated this approach by assuming a different model of cost: the number of transfers required, including the transmission of the address to the memory. Thus a single memory access, for example, if no cache were present, would require two transfers. A line access incorporating 4 transfers of data would then have a cost of 5. Under

Table 5: BUS TRAFFIC RATIO vs. LINE SIZE NON-LINEAR BUS COST

Line Size (Bytes)	4K Bytes		16K Bytes	
	Cold Start	Warm Start	Cold Start	Warm Start
4	**6.17**	**5.17**	4.22	3.84
8	6.50	5.34	4.04	**3.46**
16	7.95	5.95	**3.99**	3.71
32	13.0	8.63	4.23	3.81
64	26.5	17.1	5.66	4.86
128	63.7	36.6	9.68	7.43
256	154	96.3	25.5	17.1

this cost assumption, we computed the bus traffic, again relative to that for no cache. The results are shown in table 5. Under this cost assumption the traffic for a two-transfer line (eight bytes) is still higher for a 4K-byte cache, but only marginally so. For a 16K-byte cache it actually declines, the minimum bus traffic reached at 8-byte lines for steady-state, 16-byte lines during the start-up transient. For larger lines the increase in the bus traffic is still substantial.

We conclude that for the linear cost model, a single data transfer (4 bytes) per miss is generally best, though two transfers improves the hit ratio by roughly the same amount that it increases the bus traffic for a 4K-byte cache, double that for a 16K-byte cache. For the non-linear bus cost model, we conclude that an 8-byte line for a 4K-byte cache provides neither the lowest miss ratio, nor the lowest bus traffic, but is near enough on both counts to be the best choice. For a 16K-byte cache, 16-byte lines is probably best, though 8-byte lines yield slightly lower bus traffic under steady-state conditions.

5.5.1. Lowering the Overhead of Small Lines

Small lines are costly in that they greatly increase the overhead of the cache: an address tag and the two state bits are normally stored in the cache for each line. We reduced this overhead by splitting the notion of line into two parts:

(1) A *block* is the amount of data transferred from backing store into the cache on a read miss.

(2) A *sector* is the quantum of storage for which a tag is maintained in the cache. It is always the

Table 6a: MISS RATIO vs. SECTOR SIZE
Block Size is 4 Bytes.

Sector Size (Bytes)	4K-Byte Cache		16K-Byte Cache	
	Cold Start	Warm Start	Cold Start	Warm Start
4	**5.44**	**4.39**	**3.66**	**3.33**
8	6.28	4.74	4.00	3.41
16	7.73	5.33	4.01	3.68
32	10.9	6.82	4.15	3.72
64	15.8	9.35	4.45	3.84
128	24.4	13.7	5.32	4.08
256	30.5	17.8	8.04	4.94

Table 6b: MISS RATIO vs. SECTOR SIZE
Block Size is 8 Bytes.

Sector Size (Bytes)	4K-Byte Cache		16K-Byte Cache	
	Cold Start	Warm Start	Cold Start	Warm Start
8	**3.83**	**2.91**	**2.37**	**2.01**
16	4.69	3.27	2.38	2.19
32	6.66	4.19	2.47	2.22
64	9.73	5.80	2.68	2.31
128	15.2	8.44	3.27	2.50
256	19.3	11.2	4.95	3.09

same size as, or a power of two larger than a line.[7]

We have discovered that an effective cache can be achieved by keeping the block small but making the sector larger.

For most commercial products containing a cache, the sector is identical to the block, though in at least one [IBM74] the sector contains two blocks. The IBM System/360 Model 85 [Liptay68] included a fully associative cache with 1K-byte sectors, each containing 64 16-byte blocks.

5.5.2. The Effect of Large Sectors

The use of sectors larger than blocks means that only data from one sector in backing store can occupy any of the blocks making up a sector in the cache. There are cases where the appropriate block is empty, but other blocks in the same sector must be purged so that the new sector can be allocated. We examined this for various sizes of sectors and found that the miss ratio increased very slowly up to a point. Table 6a and 6b show how the miss ratio varies with sector size for a constant block size -- 4 bytes for table 6a, 8 bytes for table 6b. When the cache is sufficiently large (16K bytes) the miss ratio rises very slowly with increasing sector size. Under warm-start conditions, the miss ratio has increased by less than 25% when the sector size is increased from the block size to

Table 7a: BUS TRAFFIC RATIO vs. SECTOR SIZE
Reservation by Sector
Block Size is 4 Bytes.

Sector Size (Bytes)	4K-Byte Cache		16K-Byte Cache	
	Cold Start	Warm Start	Cold Start	Warm Start
4	**6.17**	**5.17**	4.22	3.84
8	6.49	5.24	4.10	**3.49**
16	7.63	5.68	**3.87**	3.56
32	10.6	7.16	3.88	3.52
64	15.3	9.78	4.15	3.67
128	23.2	14.2	4.96	3.96
256	28.8	18.7	7.61	4.95

Table 7b: BUS TRAFFIC RATIO vs. SECTOR SIZE
Reservation by Sector
Block Size is 8 Bytes.

Sector Size (Bytes)	4K-Byte Cache		16K-Byte Cache	
	Cold Start	Warm Start	Cold Start	Warm Start
8	**8.25**	**6.80**	5.11	**4.36**
16	9.71	7.30	**4.89**	4.52
32	13.5	9.17	4.93	4.49
64	19.6	12.6	5.34	4.73
128	30.3	18.1	6.51	5.17
256	38.4	24.3	9.93	6.53

[7] In this paper we use the term *line* when either (1) the block and sector are the same, or (2) a choice exists, and either *block* or *sector* may be appropriate.

128 bytes for either four- or eight-byte blocks. For a smaller cache (4K bytes), the increase in the miss ratio is more substantial, doubling for 64-byte sectors. In all cases the situation is somewhat worse during cold start.

The bus traffic decreases even more slowly (tables 7a and 7b). In fact, for the 16K-byte cache it actually *declines* as the sector size increases. For both four and eight-byte blocks the minimum bus traffic is reached with a 16-byte sector size for cold start, 8-bytes for warm start. For eight-byte blocks the bus traffic is only 19% higher for 128-byte sectors than for eight-byte sectors.

We point out, however, that an additional factor has come into play here. The write-once algorithm employed reserves sectors, not blocks. This would seem to increase greatly the traffic whenever the line is purged from the cache, but in fact the effect is small: only those blocks which have actually been modified need be written back (though this requires maintaining dirty bits for each *block*). Because of write locality, it is often possible to eliminate writes by grouping small blocks into sectors. Effectively, write-back is employed in all but one block of a sector. If a write were necessary for each block to assure exclusive access, more traffic would be generated. This effect is shown in tables 8a and 8b, where the reservation is done on a block, rather than a line, basis. With this algorithm, the bus traffic correlates very well with the miss ratio.

For any block size, clearly a better hit ratio will be achieved by associating a tag with every block, *i.e.*, making sectors and blocks the same size. For commercial machines having larger sectors than blocks, we believe it was done to reduce the size of the tag memory. However, if bus traffic is being optimized rather than hit ratio, multiple blocks per sector may be justified for better performance. Using larger sectors reduces the cost of the tag memory and, to a point, will not significantly increase the bus traffic or miss ratios.

5.6. Implications for Multiple Processors

The simulation studies reported here assumed a single processor, running a single process. They demonstrate that it is possible to reduce the bus traffic to memory by more than 90% by the introduction of an appropriate cache. This suggests that it is therefore possible either (1) to support a high-performance processor (with infrequent context switching) with memory connected by a bus of insufficient bandwidth when used in the conventional way, or (2) to support a number of processors with a single, shared memory through the use of a bus of modest speed. Our studies assumed that actual collisions resulting in cache invalidations are rare enough to be ignored for performance prediction. We have no proof that this is the case, but no evidence to the contrary.

6. Summary

Our simulations suggest that a processor with a 4K-byte cache can achieve a miss ratio below 5% while requiring only about 5% of the traffic to memory necessary if no cache is employed. A processor with a 16K cache with 8-byte lines requires even less traffic, and can achieve a miss ratio of about 2%. If 64-byte sectors are employed -- reducing the number of tags from 2048 to 256 -- the hit ratio will increase to about 2.5%, while the bus traffic will still be under 5%. This assumes no task switching, but a cache consistency algorithm which allows multiple processors to maintain private caches.

An important result is the use of the write-once algorithm to guarantee consistent data among multiple processors. We have shown that this algorithm can be implemented in a way that degrades performance only trivially (ignoring actual collisions, which are rare), and performs better than either pure write-back or write-through in many instances.

The use of small blocks with larger sectors results in an inexpensive cache which performs effectively in the absence of frequent process switches. The low bus utilization and the solution to the stale data problem make possible an environment for which this condition is met. As the sector is enlarged while holding the block size constant, the miss ratio increases some, but bus traffic may actually decline. Therefore sectors should be used as the allocation unit for reserving memory for modification even if small blocks are used for transfer of data.

Table 8a: BUS TRAFFIC RATIO vs. SECTOR SIZE
Reservation by Block
Block Size is 4 Bytes.

Sector Size (Bytes)	4K-Byte Cache		16K-Byte Cache	
	Cold Start	Warm Start	Cold Start	Warm Start
4	**6.17**	**5.17**	**4.22**	**3.84**
8	7.06	5.58	4.56	3.92
16	8.60	6.22	4.57	4.21
32	12.0	7.88	4.72	4.26
64	17.3	10.7	5.07	4.42
128	26.3	15.2	6.02	4.70
256	32.7	19.8	8.96	5.74

Table 8b: BUS TRAFFIC RATIO vs. SECTOR SIZE
Reservation by Block
Block Size is 8 Bytes.

Sector Size (Bytes)	4K-Byte Cache		16K-Byte Cache	
	Cold Start	Warm Start	Cold Start	Warm Start
8	8.25	6.80	5.11	4.36
16	10.1	7.59	5.15	4.76
32	14.2	9.66	5.33	4.85
64	20.7	13.3	5.81	5.16
128	32.0	18.8	7.09	5.65
256	40.5	25.2	10.7	7.12

The approach advocated here is appropriate only for a system containing a single logical memory. This is significant because it depends on the serialization of memory accesses to assure consistency. It has applications beyond those studied here, however. For example, the access path to memory could be via a ring network, or any other technique in which every request passes every processor. This extension seems particularly applicable for maintaining consistency for a file system or a common virtual memory being supplied to multiple processors through a common bus such as Ethernet[Metcalfe76].

Clearly there are many environments for which this model is inappropriate -- response to individual tasks may be unpredictable, for example. However, we believe that such a configuration has many potential applications and can be exploited economically if the appropriate VLSI components are designed. We have investigated the design of such components and believe that they are both feasible and well-suited for VLSI [Goodman83, Ravishankar83].

Our analysis indicates that the cache approach is reasonable for a system where bandwidth between the CPU and most of its memory is severely limited. We have demonstrated through simulation of real programs that a cache memory can be used to significantly reduce the amount of communication a processor requires. While we were interested in this for a single-chip microcomputer of the future, we have also demonstrated that such an approach is feasible for one or more currently popular commercial markets.

7. Acknowledgements

This material is based upon work supported by the National Science Foundation under Grant MCS-8202952.

We wish to thank T.-H. Yang for developing the trace facility. M.-C. Chiang did extensive programming both for the trace facility and the cache simulator. P. Vitale and T. Doyle contributed much through discussions and by commenting on an early draft of the manuscript. A. R. Pleszkun offered many helpful comments on later versions.

8. References

[Amdahl 82] C. Amdahl, *private communication*, March 82.

[Archibald 84] J. Archibald and J.-L. Baer, "An Economical Solution to the Cache Coherence Problem," *Eleventh Annual Symposium on Computer Architecture*, June 1984, pp. 355-371.

[Bell 74] J. Bell, D. Casasent, and C. G. Bell, "An investigation of alternative cache organizations," *IEEE Trans. on Computers*, Vol. C-23, No. 4, April 1974, pp. 346-351.

[Bell 78] C. G. Bell, J. C. Mudge, and J. E. McNamara, *Computer Engineering: A DEC View of Hardware Systems Design*, Digital Press, Bedford MA 01730, (1978).

[Censier 78] L. M. Censier and P. Feautrier, "A new solution to coherence problems in multicache systems," *IEEE Trans. on Computers*, Vol. C-27, No. 12, December 1978, pp. 1112-1118.

[Clark 83] D. W. Clark, "Cache Performance in the VAX-11/780," *ACM Transactions on Computer Systems*, Vol. 1, No. 1, (February 1983), pp. 24-37.

[Easton 78] M. C. Easton and R. Fagin, "Cold-start vs. warm-start miss ratios," *CACM*, Vol. 21, No. 10, October 1978, pp. 866-872.

[Frank 84] S. J. Frank, "Tightly coupled multiprocessor system speeds memory-access times," *Electronics*, Vol. 57, No. 1, (January 12, 1984), pp. 164-169.

[Goodman 83] J. R. Goodman, "Using Cache Memory to Reduce Processor/Memory Traffic," *Tenth Annual Symposium on Computer Architecture*, June 1983, pp. 124-131.

[Grosch 53] H. A. Grosch, "High Speed Arithmetic: the Digital Computer as a Research Tool," *Journal of the Optical Society of America*, Vol. 43, No. 4, (April 1953).

[Hoogendoorn 77] C. H. Hoogendoorn, "Reduction of memory interference in multiprocessor systems," *Proc. 4th Annual Symp. Comput. Arch.*, 1977, pp. 179-183.

[IBM 74] "System/370 model 155 theory of operation/diagrams manual (Volume 5): buffer control unit," IBM System Products Division, Poughkeepsie, N.Y., 1974.

[IBM 76] "System/370 model 168 theory of operation/diagrams manual (Volume 1)," Document No. SY22-6931-3, IBM System Products Division, Poughkeepsie, N.Y., 1976.

[Intel] "Multibus-II Bus Architecture Specification Handbook," Pub. No. 146077-B, Intel Corporation, Santa Clara, CA.

[Kaplan 73] K. R. Kaplan and R. O. Winder, "Cache-based computer systems," *Computer*, March 1973, pp. 30-36.

[Isaak 82] J. Isaak, "Squeezing the Most Out of the 68000," *Mini-Micro Systems*, October 1982, (pp. 193-202).

[Knight 66] J. R. Knight, "Changes in computer performance," *Datamation*, Vol. 12, No. 9, September 1966, pp. 40-54.

[Lindsay 81] D. C. Lindsay, "Cache Memory for Microprocessors," *Computer Architecture News, ACM - SIGARCH*, Vol. 9, No. 5, (August 1981), pp. 6-13.

[Liptay 68] "J. S. Liptay, "Structural aspects of the System/360 Model 85, Part II: the cache," *IBM Syst. J.*, Vol. 7, No. 1, 1968, pp. 15-21.

[Metcalfe 76] R. M. Metcalfe and D. R. Boggs, "Ethernet: Distributed Packet Switching for Local Computer Networks," *Comm. ACM*, (July 1976), pp. 395-404.

[Norton 82] R. L. Norton and J. L. Abraham, "Using write back cache to improve performance of multiuser multiprocessors," *1982 Int. Conf. on Par. Proc.*, IEEE cat. no. 82CH1794-7, 1982, pp. 326-331.

[Patel 82] J. H. Patel, "Analysis of multiprocessor with private cache memories," *IEEE Trans. on Computers*, Vol. C-31, No. 4, April 1982, pp. 296-304.

[Pier 83] K. A. Pier, "A retrospective on the Dorado, a high-performance personal computer," *Tenth Annual Symposium on Computer Architecture*, June 1983, pp. 252-269.

[Rao 78] G. S. Rao, "Performance Analysis of Cache Memories," *Journal of the ACM*, Vol. 25, No. 7, July 1978, pp. 378-395.

[Ravishankar 83] C. V. Ravishankar and J. R. Goodman, "Cache Implementation for Multiple Microprocessors," *Proceedings Spring COMPCON 83*, (February 1983), pp. 346-350.

[Siewiorek 82] D. P. Siewiorek, C. G. Bell, and A. Newell, *Computer Structures: Principles and Examples*, McGraw-Hill, New York, pp. 889-891. 1982.

[Smith 82] A. J. Smith, "Cache Memories," *Computing Surveys*, Vol. 15, No. 3, (September 1982), pp. 473-530.

[Smith 83] J. E. Smith and J. R. Goodman, "A study of instruction cache organizations and replacement policies," *Tenth Annual Symposium on Computer Architecture*, June 1983, pp. 132-137.

[Smith 85] J. E. Smith and J. R. Goodman, "Instruction Cache Replacement Policies and Organizations," *IEEE Trans. on Computers*, Vol. 34, No. 3, (March 1985), pp. ??-??.

[Solomon 66] M. B. Solomon, Jr., "Economies of Scale and the IBM System/360," *CACM*, Vol. 9, No. 6, June 1966, pp. 435-440.

[Tang 76] C. K. Tang, "Cache system design in the tightly coupled multiprocessor system," *AFIPS Proc., NCC*, Vol. 45, pp. 749-753, 1976.

[TI 84] "MOS Memory Data Book," Texas Instruments, P.O. Box 225012, Dallas, Texas 75165, (1984).

[Tredennick 82] N. Tredennick, "The IBM micro/370 project," public lecture for *Distinguished Lecturer Series*, Computer Sciences Department, University of Wisconsin-Madison, March 31, 1982.

[Widdoes 79] L. C. Widdoes, "S-1 Multiprocessor architecture (MULT-2)," *1979 Annual Report -- the S-1 Project, Volume I: Architecture*, Lawrence Livermore Laboratories, Tech. Report UCID 18619, 1979.

[Yen 82] W. C. Yen and K. S. Fu, "Coherence problem in a multicache system," *1982 Int. Conf. on Par. Proc.*, IEEE cat. no. 82CH1794-7, 1982, pp. 332-339.