USING CACHE MEMORY TO REDUCE PROCESSOR-MEMORY TRAFFIC

James R. Goodman

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

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 indeed reduce traffic to memory greatly, and introduce an elegant solution to the cache coherency problem.

1. Introduction

Because there are straightforward ways to construct powerful, cost-effective systems using random access memories and single-chip microprocessors, semiconductor technology has, until now, had the greatest impact through these components. High-performance processors, however, are still beyond the capability of a single-chip implementation and are not easily partitioned in a way which can effectively exploit the technology and economics of VLSI. An interesting phenomenon has occurred in the previous decade as a result of this disparity. Memory costs have dropped radically and consistently for computer systems of all sizes. While the component cost of a CPU (single-chip implementations excluded) has declined significantly over the same period, the reduction has been less dramatic. A result is that the amount of memory thought to be appropriate for a given speed processor has grown dramatically in recent years. Today small minicomputers have memory as large as that of the most expensive machines of a decade ago.

The impact of VLSI has been very different in microprocessor applications. Here memory is still regarded as an expensive component in the system, and those familiar primarily with a minicomputer or mainframe environment are often scornful of the trouble to which microprocessor users go to conserve memory. The reason, of course, is that even the small memory in a microprocessor is a much larger portion of the total system cost than the much larger memory on a typical main frame system. This results from the fact that memory and processors are implemented in the same technology.

1.1. A Super CPU

With the advances to VLSI occurring now and continuing over the next few years, it will become possible to fabricate circuits that are one to two orders of magnitude more complex than currently available microprocessors. It will soon be possible to fabricate an extremely high-performance CPU on a single chip. If the entire chip is devoted to the CPU, however, it is not a good idea. Extrapolating historical trends to predict future component densities, we might expect that within a few years we should be able to purchase a single-chip processor containing at least ten times as many transistors as occur in, say, the MC68000. 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 for predicting improvements within a single technology. Siewiorek in fact suggests that it holds "by definition."

Assuming $g = 1.5$ and using processor-memory bandwidth as our measure of performance, 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 10 MHz, accesses data from memory at a maximum rate of 5 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 the 68000, has stated that all modern microprocessors – the 68000

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

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. With the ability to fabricate chips containing one to two million transistors, it should be possible – using only a portion of the chip – to build a processor significantly more powerful than any currently available single-chip CPU. While devoting the entire chip to the CPU could result in a still 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 were on-chip, it might actually perform as fast as the more powerful processor.

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 probably use substantially more than that. Clearly 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.

1.3. Cache Memory

The use of cache memory, however, has often aggravated the bandwidth problem rather than reduce it. Smith [Smith82] says that optimizing the design has four general aspects:

(1) maximizing the hit ratio,

(2) minimizing the access time to data in the cache,

(3) minimizing the delay due to a miss, and

(4) minimizing the overheads of updating main memory, maintaining multicache consistency, etc.

The result is often a larger burst bandwidth requirement from main storage to the cache than would be necessary without a cache. For example, the cache on the IBM System/370 model 168, is capable of receiving data from main memory at a rate of 100 megabytes per second [IBM76]. It supplies data to the CPU at less than 1/3 that rate. The reason is that to exploit the spatial locality in memory references, the data transferred from backing store into the cache is fetched in large blocks, resulting in requirements of very high bandwidth bursts of data. We have 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 design of cache memory for mini-computers demanded greater concern for bus bandwidth. The designers of the PDP-11 models 60 and 70 clearly recognized that small block sizes were necessary to keep main memory traffic to a minimum [Bell78].

Lowering the bandwidth from backing store to the cache can be accomplished in one of two ways:

(1) small blocks of data are brought from backing store to the cache, or

(2) long delays occur 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.

We therefore have explored the effectiveness of a cache which exploits primarily or exclusively temporal locality, i.e., the blocks fetched from backing store are only the size needed by the CPU (or possibly slightly larger). In considering ways to evaluate this strategy, we identified a commercial environment that contained many of the same constraints and seemed amenable to the same kinds of solutions. This environment is the marketplace of the single-board computer running on a standard bus such as Multibus[2] or Versabus.[3] We have chosen to study this environment in an attempt to gain insight into the original, general scheme.

2. The Single Board Computer Application

A single board computer typically contains a microprocessor and a substantial amount of memory, though small enough that it must be used carefully. If needed, access to additional random access memory is through the bus, which is designed for generality and simplicity, not for high performance. Multibus, in particular, was defined in the early 70's to offer an inexpensive means of communication among a variety of subsystems. Although originally introduced by Intel Corporation, it has found wide acceptance, having been proposed – in a slightly modified form – as the IEEE P796 bus standard [IEEE80]. Currently, several hundred vendors offer Multibus-compatible cards.

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 could be increased by increasing the number of pins, and by modifying the protocol. Its broad popularity and the availability of components to implement its protocol mean, 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.

We decided to determine if a cache memory system could be implemented effectively in the Multibus environment. To that end we have designed a cache to be used with a current-generation microprocessor. In addition, we have done extensive simulation of cache performance, driven by memory trace data. We have identified a new component which is particularly suited for VLSI implementation and have demonstrated its feasibility by designing it [Ravishankar83]. This component, which implements the tag memory for a dynamic RAM cache intended for a microprocessor, is similar in many respects to the recently announced TMS 2150 [TI82].

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 has much in common with ours, 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, considerable effort is expended guaranteeing that the program running is primarily resident on-board. This approach is viable for a static partitioning of tasks. Results to date have been much less satisfactory, however, for the more general situation where a number of processors are dynamically allocated. (For efficiency reasons it also precludes the use of shared code segments).

[2]Multibus is a trademark of Intel Corporation.

[3]Versabus is a trademark of Motorola.

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 cache is not appropriate on the CRAY-1. We believe that freeing the programmer from concern about memory allocation is essential where programmer productivity is critical.

2.1. A Single-Board Computer with Cache

To evaluate our approach, we proposed 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. Thus we picked an important problem in its own right: Can we build a cache that works with a Multibus system supporting multiple processors? In particular, how many processors can we support running in parallel on Multibus? We believe that a system which could reasonably support five to 10 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 analyses [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 reasonably well. We allow individual processors to sit idle periodically rather than tie up the bus fetching data which they might not use. This implies that the cache stale data problem must be solved effectively. We present a new solution in section 3.

2.2. Switching Contexts

Where bus bandwidth is limited, a task switch is a major disturbance, since the cache must effectively be reloaded at this time. The processor is momentarily reduced to accesses at the rate at which the bus can supply them. While this problem seems unavoidable, it need not be serious if task switching is minimized. We are providing an environment which allows many processors to work out of a single monolithic memory in parallel. If more parallel tasks are required, more processors can be used. We point out that the current Multibus alternative is to move the program into local memory, an operation which also swamps the bus. The task switch merely makes this operation implicit, and avoids bringing across the bus data which are never actually used. Writing the old data out is also no worse than the alternative, since we only write that which has been changed and which has not been already purged.

There may be certain cases – an interrupt handling program, for example – where a particular program does not flush the cache, but uses only a small portion of it. Provisions could be made to allow such a program to be locked in the cache. Alternatively, a separate cache might be provided for such a program Our studies indicate that a relatively small cache can be effective for a single program, so it may be possible to keep separate caches around for individual processes if the number is small. We would suggest taking this one step further and providing an additional processor for each cache. An interesting question then arises as to the cost of dynamically assigning processes to processors. Our proposal allows this assignment, though clearly at some performance penalty.

3. 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 processors. In the single-chip processor or single-board computer environments, it has the added weakness that it requires a number of extra I/O pins.

An alternative approach, implemented in C.mmp [Hoogendoorn77] and proposed by Norton [Norton82]. is to require the operating system to recognize when inconsistencies might occur and take steps to prevent them under those circumstances. 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 blocks in this way to prevent inconsistency. Individual blocks are 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.

Our approach has much in common with the third approach, but allows the critical section information to be distributed among the caches, where it already resides. In addition, we use the normal read and write operations, with no tag bits in main memory, to accomplish the synchronization. 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. We call our scheme *write-once*.

3.1. Write-Through or Write-Back?

While the choice between write-through (also known as store-through) and write-back (also known as store-back or copy-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. For typical read-to-write and hit ratios and when task switching is infrequent, our simulations have given strong evidence 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.

3.2. A New Write Strategy: Write-Once

We propose a new write strategy which solves the stale data problem and produces minimal bus traffic. The replacement technique requires the following structure. Associated with each block in the cache are two bits defining one of four states for the associated data:

Invalid There is no data in the block.

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

Reserved The data in the block 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 block 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.

Write-once requires rapid access to the address tags and state bit pairs concurrently with accesses to the address tags by the CPU. 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. This is not a large cost, since a single chip design of this part of the cache – using present technology – is quite feasible. Further, we have discovered a way to reduce substantially the number of tags required. In addition, the same chip type could be used for both instances. This is a natural way to partition the cache in VLSI because it results in a maximal logic-to-pin ratio. We have designed and submitted for fabrication such a chip [Ravishankar83].

The two copies always contain exactly the same address data, because they are always written simultaneously. While one unit is used in the conventional way to support accesses by the CPU, a second monitors all accesses to memory via the Multibus. For each such operation, it checks for the address in the local cache. If a match is found on a write operation, it notifies the cache controller, and the appropriate block in the cache is marked *invalid*. If a match is found on a read operation, nothing is done unless the block has been modified, i.e., its state is *reserved* or *dirty*. If it is just *reserved*, the state is changed to *valid*. If it is *dirty*, the local systems 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. Initially write-through is employed. However, an additional goal is achieved upon writing. All other caches are purged of the block being written, so the cache writing through the bus now is guaranteed the only copy except for backing store. It is so identified by being marked *reserved*. If it is purged at this point, no write is necessary to backing store, so this is essentially write-through. If another write occurs, the block is marked *dirty*. Now write-back is employed and, on purging, the data must be rewritten to backing store.

[4]There is a mechanism in Multibus which allows this capability. Unfortunately, it is rarely used, not well-defined, and requires that local caches respond very rapidly. Versabus has a much cleaner mechanism by which this end can be accomplished.

Write-once has the desirable feature that units accessing backing store need not have a cache, and need not know whether others do or not. A cache is responsible for maintaining consistency exactly for those cases where it might create a violation, i.e., whenever it writes to a location. 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 block.

4. Simulation

We designed a cache memory system to work on Multibus. To validate our design before building it we did extensive simulation using memory trace data. To date we have performed extensive simulations for six traces, all running under UNIX:[5]

EDC The UNIX editor *ed* running a script.

ROFFAS The old UNIX text processor program *roff*.

TRACE The program, written in assembly language, which generated the above traces for the PDP-11.

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.

The first three traces are for a PDP-11, while the latter three are for a VAX. While the PDP-11 does not run on Multibus, its instruction set is similar to many microprocessors which do, and the programs used for tracing were of the kind we envision for such a system. The PDP-11 is similar in many ways to the MC68000, and has in common with the 8086 a limited addressing capability.

While the VAX also does not run on Multibus, it is an example of a modern instruction set and, therefore is a reasonable example of the kind of processor likely to appear in a single-chip CPU in the future. It also has a larger address space which, as shown in section 4.3, is significant. We are actually using virtual addresses, but all of the programs we ran are small enough to fit into main memory. Since we are tracing only a single process, we conclude that there is no significant difference between virtual and real addresses.

In addition to cache parameters, miss ratios vary greatly depending on the program running. For the 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 three 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 actually the average of three programs, each running alone.

4.1. 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 block size is large. Under write-back, when a dirty block is purged, the entire block 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 blocks are very large, or (2) when the cache size is very small.

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

Write-once results in bus traffic roughly equal to the better of the two. We have found cache parameters for which it actually performs better on the average than either write-through or write-back for a number of programs. This was a surprising result, since write-once was developed to assure coherency, not to minimize bus traffic. The replacement scheme outperforms both write-through and write-back whenever the total number of sets is about 16. For example, for a 4-way set associative, 2048-byte cache with a block size of 32 bytes, the average bus traffic for three PDP-11 programs for which we have traces was 30.768% for write-through, 17.55% for write-back, and 17.38% for write-once.

4.2. Cold Start vs. Warm Start

An important consideration in determining cache hit ratio and bus traffic is the cold start period known as the *lifetime function* [Easton78], during which time many misses occur because the cache is empty. This is defined as the period until as many cache misses have occurred as the total number of blocks in the cache. This initial burst of misses is amortized over all 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. Thus an important assumption in traditional cache evaluation is the frequency of task switching. We have argued that task switching must be very infrequent in our system. Thus we can more nearly approach in practice the warm start hit ratios, and thus it is appropriate to use very long traces of a single program, and assume a warm start. We did that initially, using the full length of the PDP-11 traces available to us (1,256,570 memory accesses). We noted, however, that for even much shorter traces than we were running, there was little difference between warm start and cold start statistics. Since cold start statistics are easier to generate, we normally used them. Unless stated otherwise, our results are from cold start, but at least 10 times the lifetime function in total length.

4.3. Cache Size

In general, we were surprised at the effectiveness of a small cache. For the PDP-11 traces with a cache of 2K bytes or larger, we discovered that essentially no misses occurred after the cold start period. These are not trivial programs, but were run on a machine which has only 64K bytes for both instructions and data. The programs are very frugal in their use of memory, and the entire working set apparently can fit in the cache.

The VAX traces do not exhibit the same locality observed with the PDP-11, and a 64K-byte cache was not large enough to contain the entire working set of the program. This may be a result of the larger address space available, the more complex instruction set, or more complex programs. In all cases the programs were spread out over a much larger memory space than for the PDP-11 traces. For this comparison we used a small block size of 4 bytes. This may have had a greater impact on the VAX than on PDP-11 traces.

We found that reducing the size of the cache (below 4K bytes for the PDP-11) increased the miss ratio and the bus traffic – in general the two correlate well with respect to this parameter. Fig. 1 shows the average miss ratio and bus traffic as a function of total cache size for the PDP-11 traces. For this and all results given, the miss ratio includes writes. The bus traffic is given as a percent of the number of accesses that would be required if no cache were present. Fig. 2 shows the same data for the VAX traces.

4.4. Block Size

Our cache design incorporates extremely small blocks, depending heavily on temporal locality. Easton and Fagin [Easton78] claim that page size and miss ratios are independent for warm start, but highly dependent for cold start. If true, this can be explained by the observation that hits in the cache on cold start depend heavily on spatial locality, while temporal locality provides many hits when it is warm. Spatial locality, however, is strongly correlated to block size, being directly proportional in the extreme case of strictly sequential memory accesses. Our simulations partially confirm Easton's observation. In particular, we found that, as block size is increased, miss ratios generally decline up to a point, then increase for either warm or cold starts. However, for small block sizes, the warm start miss ratio is marginally lower than for the cold start case, while for large block sizes, the two numbers are nearly identical. See figs. 3-7. This is encouraging since we have argued for restricted task switches: our environment is more that of a warm start than is the traditional environment. In many simulations we were able to get very high hit ratios once the cold start period ended. For small blocks transferred, however, this period (the lifetime function) is longer. Our simulations show very clearly that reducing the block size down to a single transfer across the bus dramatically decreases the hit ratio, particularly for cold starts, but also decreases bus traffic significantly. In general, we observed that increasing the transfer block size from one bus cycle to two typically decreases the miss ratio by 30 to 50%, while increasing the bus traffic by 10 to 20%. This relation holds for the first two doublings. These results, e.g., fig. 3, are relatively more pessimistic for small block sizes than those reported by Strecker in [Bell78].

We have made the assumption that access time is related linearly to block 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, and arbitration is overlapped with bus operations. For a single-chip implementation, it would almost certainly be worthwhile to provide the capability for efficient multiple transfers over a set of wires into the processor. This has not been incorporated in our analysis, but will undoubtedly suggest a somewhat larger transfer block size.

4.4.1. Lowering the Overhead of Small Blocks

Small blocks 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 block transferred. We reduced this overhead by splitting the notion of block into two parts:

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

(2) The *address block* is the quantum of storage for which a tag is maintained in the cache. It is always a power of two larger than a transfer block. An effective cache can be implemented by keeping the transfer block small but making the address block larger.

For most commercial products containing a cache, the address block size is the same as the transfer block size, though we know of one example [IBM74] where the address block contained two transfer blocks. The IBM System/360 Model 85 [Liptay68] in fact is a special case of this, *viz.*, a direct-mapped cache, where the Model 85 "sector," consisting of 1K bytes, corresponds to our address block. Each sector contains 16 transfer blocks, which were called simply "blocks."

4.4.2. The Effect of Large Address Blocks

The use of address blocks larger than transfer blocks means that only data from one address block in

backing store can occupy any of the blocks making up an address block in the cache. There are cases where the appropriate transfer block is empty, but other transfer blocks in the same address block must be purged so that the new address block can be allocated. We examined this for various sizes of address blocks and found that the miss ratio increased very slowly up to a point. For the situation shown in fig. 7, the miss ratio had only risen by about 30% when the address block contained 64 bytes for the PDP-11 traces. That point was reached for the VAX traces when it contained 32-byte address blocks.

We predicted that the bus traffic would correlate well with miss ratio with respect to this parameter. To our surprise, the bus traffic actually *declined* initially as we increased the address block size. The decline was small, but consistent for the PDP-11 trace tapes, eventually climbing over the base line when the address block was 16 or 32 transfer blocks. The phenomenon was smaller, but discernible for the VAX traces as well, though in all cases the bus traffic started increasing sooner. This situation is shown in fig. 8.

We initially suspected that our simulation might be faulty. That was not the case, and eventually we were able to explain it and verify it. The write-once algorithm requires a bus operation whenever a block is modified initially (set to *reserved.)* However, reservations could be made on the basis of either transfer blocks or address blocks. We had put the choice into the simulator, but had not experimented with it, reserving at the address block level. This in fact reduces the number of bus writes necessary for reservation because of spatial locality of writes: an address block already *reserved* need only be marked *dirty* when any transfer block within it is modified. This would seem to increase greatly the traffic whenever the block is purged from the cache, but in fact the effect is small: only those transfer blocks which have actually been modified need be written back.

We demonstrated that this was indeed responsible for the behavior noted by changing the reservations to the transfer block level. The simulation results then exhibited the originally predicted behavior, correlating closely with the miss ratio.

We conclude that minimum bus traffic is generated with minimum transfer block sizes. The miss ratio may be substantially improved by using slightly larger transfer blocks, in which case bus traffic does not increase greatly. Using larger address blocks reduces the cost of the tag memory considerably. It initially has only a minor effect on miss ratio, which is more than offset by the savings in writes due to the more efficient reservation of modified blocks

4.5. Other Design Aspects Studied

4.5.1. Write Allocation

Write allocation, also known as *fetch on write*, means that a block 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 blocks. This was true using both the measures of miss ratio and bus traffic. In all results presented, write allocation was employed.

4.5.2. 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 have little new to report, i.e., fully associative cache is the best, but 2-way set associative is not much worse, and 4-way set associative is somewhere in between. (But see [Smith83]). We had hoped to find that a high degree of associativity would improve performance, because such an organization is much more feasible in the VLSI domain, but results were negative. For results reported here we have assumed a 4-way, set associative cache.

4.5.3. Replacement Algorithm

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

4.5.4. Bus Width

The width of the data paths between units is an important parameter in that it is closely related to bandwidth. We have the capability to specify the bus width both from backing store to cache and from cache to CPU. For the purposes of this study, we have assumed in all cases that the VAX memory supplies 4 bytes to the cache in one bus cycle, while the PDP-11 memory supplies 2 bytes. An 8-byte transfer therefore is counted as two cycles for the VAX and 4 for the PDP-11.

We assumed that the cache supplied one word – 16 bits for the PDP-11 and 32 bits for the VAX – to the CPU on each request. However, the intelligence of the processor determines how often the same word must be fetched. The trace tapes contain all memory references. We filtered these with the assumption that on instruction fetches the same word would not be fetched without an intervening instruction fetch. No filtering was done on data fetches.

5. Summary

Our simulations indicate that a single board computer with a 4K-byte cache can perform reasonably well with less than 10% of the accesses required to its primary memory without a cache. The PDP-11 traces suggest a number as low as 3%. While the VAX numbers are higher, additional declines will be experienced by increasing the size of the cache beyond 4K bytes.

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 transfer block sizes can be coupled with large address blocks to build 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. Even though the miss ratio increases, bus traffic initially declines as the address block is enlarged, holding the transfer block constant. Therefore larger address blocks should be used for reserving memory for modification even if small blocks are used for transfer of data.

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 at the for maintaining consistency for a file system or a common virtual memory being supplied to

multiple processors through a common bus such as Ethernet.

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 [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.

6. Acknowledgements

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

We thank Dr. A. J. Smith for providing the PDP-11 trace tapes upon which much of our early work depended. We also wish to thank T.-H. Yang for developing the VAX trace facility. P. Vitale and T. Doyle contributed much through discussions and by commenting on an early draft of the manuscript.

7. References

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

[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. Bell, J. Judge, J. McNamara, *Computer engineering: a DEC view of hardware system design*, Digital Press, Bedford, Mass., 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.

[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.

[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.

[IEEE 80] "Proposed microcomputer system bus standard (P796 bus)," *IEEE Computer Society Subcommittee Microcomputer System Bus Group*, October 1980.

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

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

[Lindsay 81] "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.

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

[Rao 78] G. S. Rao, "Performance Analysis of Cache Memories," *Journal of the ACM*, Vol. 25, July 1978, pp. 378-395.

[Ravishankar 83] C. V. Ravishankar and J. Goodman, "Cache implementation for multiple microprocessors," *Digest of Papers, Spring COMPCON 83*, IEEE Computer Society Press, March 1983.

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

[Smith 82] A. J. Smith, "Cache memories," *Computing Surveys*, Vol. 14, 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.

[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 82] *Texas Instruments MOS Memory Data Book*, Texas Instruments, Inc., Memory Division, Houston, Texas, pp. 106-111, 1982.

[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 1: 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.

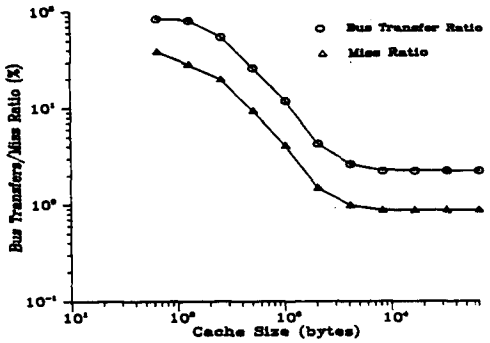

Fig. 1. Bus Transfer and Miss Ratios vs. Cache Size; blocks are 4 bytes; PDP-11 traces. The bus transfer ratio is the number of transfers between cache and main store relative to those necessary if there were no cache.

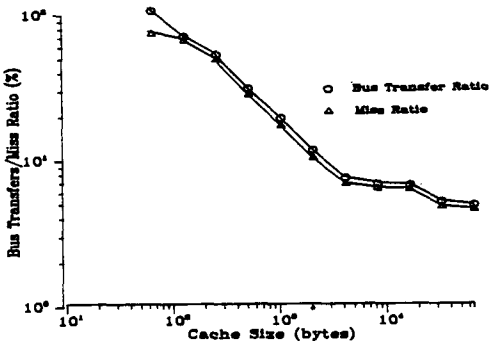

Fig. 2. Bus Transfer and Miss Ratios vs. Cache Size; 4-byte blocks; VAX-11 traces.

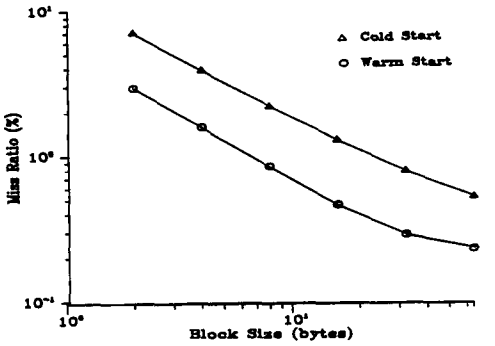

Fig. 3. Miss Ratio vs. Block Size for warm and cold starts; PDP-11 traces.

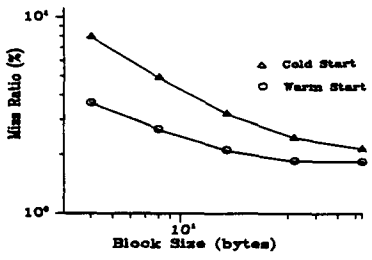

Fig. 4. Miss Ratio vs. Block Size for warm and cold starts; VAX-11 traces.

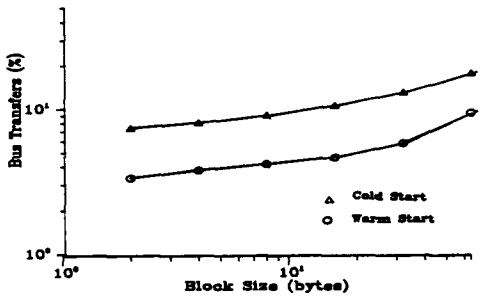

Fig. 5. Bus Transfer Ratio vs. Block Size for warm and cold starts; PDP-11 traces.

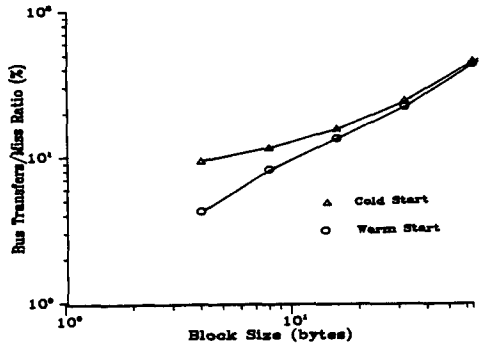

Fig. 6. Bus Transfer Ratio vs. Block Size for warm and cold starts; VAX-11 traces.

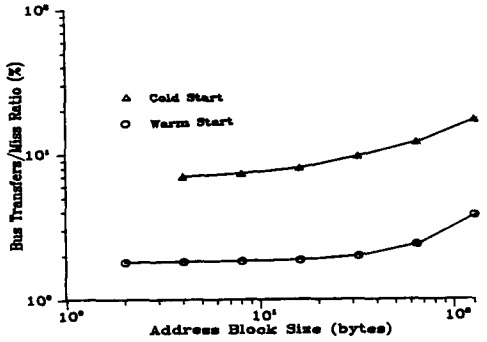

Fig. 7. Miss ratio vs. Address Block Size for warm and cold starts.

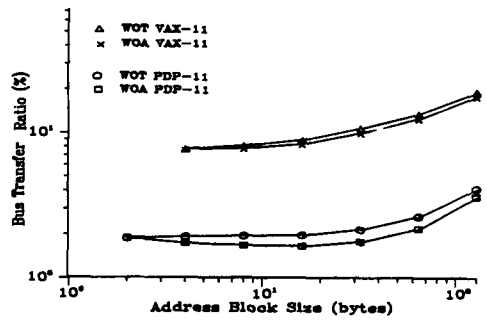

Fig. 8. Bus Transfer Ratio vs. Address Block Size for warm and cold starts; WOA: address blocks are reserved. WOT: transfer blocks are reserved.