The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor

James R. Goodman
and
Philip J. Woest

Computer Sciences Technical Report #766

April 1988

The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor

James R. Goodman and **Philip J. Woest**

Computer Sciences Department
University of Wisconsin-Madison

Abstract–The *Wisconsin Multicube*, is a large-scale, shared-memory multiprocessor architecture that employs a snooping cache protocol over a grid of buses. Each processor has a conventional (SRAM) cache optimized to minimize memory latency and a large (DRAM) snooping cache optimized to reduce bus traffic and to maintain consistency. The large snooping cache should guarantee that nearly all the traffic on the buses will be generated by I/O and accesses to shared data.

The programmer's view of the system is like a multi -- a set of processors having access to a common shared memory with no notion of geographical locality. Thus writing software, including the operating system, should be a straightforward extension of those techniques being developed for multis.

The interconnection topology allows for a cache-coherent protocol for which most bus requests can be satisfied with no more than twice the number of bus operations required of a single-bus multi. The total symmetry guarantees that there are no topology-induced bottlenecks. The total bus bandwidth grows in proportion to the product of the number of processors and the average path length.

The proposed architecture is an example of a new class of interconnection topologies -- the *Multicube* -- which consists of $N=n^k$ processors, where each processor is connected to k buses and each bus is connected to n processors. The hypercube is a special case where $n=2$. The Wisconsin Multicube is a two-dimensional Multicube ($k=2$), where n scales to about 32, resulting in a proposed system of over 1,000 processors.

1. Introduction

Shared-memory multiprocessors represent an effective means of providing substantial processing power at a reduced cost. Cache coherent multiprocessors known as *multis*, [Bell85] have the important advantage that cache coherency can be maintained efficiently by hardware that monitors the bus traffic, providing an elegant, simple view of memory to the programmer. Since the coherency mechanism generally relies on the ability of every cache controller to observe every bus transaction, multis are restricted to a single-bus implementation. Thus, this class of multiprocessors is limited to some tens of processors.

Numerous large-scale multiprocessors based on multi-stage interconnection networks have been studied [GGKM83, GKLS83, Butt85, PBGH85, Lund87]. However, since there are no efficient mechanisms known for maintaining hardware cache consistency among large-scale multiprocessors, these architectures generally do not allow shared data blocks to migrate from global shared memory to local memories or caches. This requires system programmers and/or compilers to be concerned with explicitly mapping processes and data onto the various processors and memories for efficient execution.

The hypercube architecture offers an attractive, symmetric topology that can scale to a very large number of processors. Unfortunately, the architectures based on this topology to date have no notion of shared memory, necessitating communication through messages and again requiring explicit mapping of processes and data onto processors and memory. Also, for a large number of processors, the number of intermediate nodes traversed is substantial ($O(lg_2 N)$) for arbitrary paths, necessitating careful analysis of algorithms and assignments for good performance.

This paper presents a design for a new, large-scale multiprocessor architecture -- the *Wisconsin Multicube* -- which employs a snooping cache protocol over a grid of buses to produce an image of shared memory like that provided by the multi. Memory requests which produce cache misses typically require no more than twice as many bus operations as that required for a single bus multiprocessor. Furthermore, the symmetry of the organization allows bus traffic to be distributed uniformly across the buses, reducing the probability of bottlenecks. The architecture provides for a multi-level cache structure: a conventional (SRAM) cache for reducing memory latency, and a very large (DRAM) cache for minimizing bus traffic.

The Wisconsin Multicube is a very attractive architecture for developing parallel applications. While providing a view of a single shared memory to the programmer, it imposes no notion of geographical locality. Furthermore, this architecture may be viewed as a collection of conventional multis connected by orthogonal sets of buses

which transparently extend the snooping cache protocol to higher dimensions. This ensures that applications developed for multis can be easily converted to this architecture. Thus, the Wisconsin Multicube is intended to be a general purpose multiprocessor which supports a large range of applications, such as high-transaction database systems, large-scale simulation models, and artificial intelligence applications, as well as a host of numerical methods.

Currently, we are designing the memory hierarchy (the caches and controllers, the cache coherency protocol and the main memory), and other aspects of the system. With little published data on the memory reference behavior of parallel programs it is clear that further study of this area will be necessary to complete the design. Even so, initial results using an approximate mean-value analysis have been used to investigate effective execution speed for various numbers of processors and rates of shared data access, methods for reducing bus latency, the effects of invalidation traffic, and other factors [LeVe88]. Some results from these studies are reported here.

The remainder of the paper is organized as follows. Section 2 describes the architecture of the Wisconsin Multicube and section 3 describes its cache coherency protocol. Section 4 discusses efficient synchronization primitive. Section 5 discusses a number of other important design issues and modeling results. Section 6 introduces the general Multicube topology and discusses the scalability of the Wisconsin Multicube. Section 7 contains a summary. A formal description of the cache consistency protocol is contained in Appendix A.

2. System Architecture

The architecture employs a snooping cache system over a grid of buses, as shown in figure 1. Each processor is connected to a multi-level cache. The first level cache, referred to as the processor cache,[1] is a high-performance (SRAM) cache designed with the traditional goal of minimizing memory latency. A second level cache, referred to as the snooping cache, is a very large (minimum size: 64 DRAMs) cache designed to minimize bus traffic [Good83]. Each snooping cache monitors two buses, a row bus and a column bus, in order to maintain data consistency among the snooping caches. Consistency between the two cache levels is maintained by using a write-through strategy to assure that the processor cache is always a strict subset of the snooping cache [BaWa87].

The proposed cache structure should reduce the bus traffic to the point that nearly all operations are either accesses to true shared data, or they are true I/O. A write-back cache-coherency protocol will eliminate most bus traffic due to writes. Similarly, the large snooping cache should obviate nearly all bus traffic from

[1]This cache is external to the processor, which likely will include an on-chip cache as well.

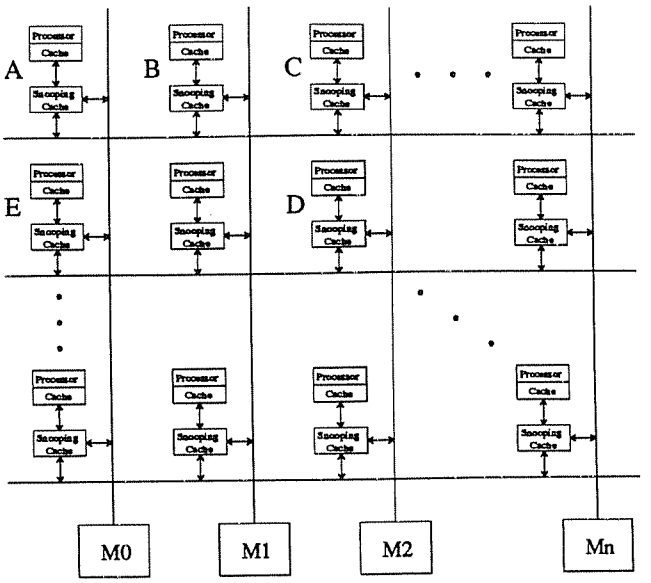

Figure 1: The Wisconsin Multicube

conventional cache misses on reads. The size of this snooping cache is comparable to main memory on most current machines, so for a comparable workload, conventional cache miss traffic should be comparable to page traffic in current systems.[2]

A computer system of this magnitude has massive, varied I/O requirements. While providing high bandwidth to memory, the cache consistency mechanism must not be broken, nor can there be interference with the processor demands upon the memory system. The solution to these problems is to attach the I/O directly to some or all of the processors, with DMA routed through the processor's snooping cache. At the interconnection level, I/O is then treated as any other processor request for shared data, thus providing great flexibility in I/O capabilities, while avoiding much of the double writing normally associated with DMA on conventional bus systems. In the proposed machine, I/O data may never actually be written to memory, but be read directly across the bus into the cache of the processor requesting it.

Many aspects must be considered in the design of the proposed machine. The most significant component is the design of an efficient protocol to extend the snooping bus consistency mechanism to a grid of buses. A preliminary design of the protocol is discussed below. Additional design issues, especially those related to synchronization and reducing bus latency, are presented in Sections 4 and 5.

[2]Of course, the enormous total memory and computing power of this machine will permit the analysis of problems of unprecedented size. How the memory of such a system might be used can only be a matter of speculation.

3. A Multiple Bus Cache Consistency Protocol

In a multi, bus operations require certain actions to be performed by one or all of the system's snooping caches. Read requests are satisfied by main memory, or possibly by some other processor's cache. In the Wisconsin Multicube, each row of processors is similar to a multi. To connect the rows, column buses are added so that each processor is connected to two buses. Main memory is located on the columns, interleaved by lines or pages. Thus, each line of memory has a *home column*, that is, a column bus through which it can be accessed from main memory.

In this scheme a line is always in one of two global states.[3] The line may be in the *unmodified* state, in which case the value stored in memory is correct, but additional copies may be in one or more caches throughout the system. Or the line may be in the *modified* state, in which case the value in memory is stale and the correct value resides in exactly one cache. With respect to a particular cache, a line may be in one of three local modes: *shared* (global unmodified state), *modified* (global modified state with the line present in only this cache), or *invalid*. A line that is invalid, but was recently contained in the cache, may be acquired (*snarfed*) in shared mode as it passes by, but only if the line is in global state unmodified.

This organization allows a controller to issue a request on its row bus that will require at most twice the number of bus operations as in a conventional multicomputer. However, there are two main differences, both a result of the inability of every controller to monitor every bus operation. First, a write to a line that may be shared requires an invalidation of all of the copies. This results in a broadcast operation which requires a single column bus operation followed by a bus operation on every row bus. Second, when a request is made for a line that has been modified, it is necessary to find which snooping cache currently has that line.

The latter problem can be solved using two auxiliary memory structures. First, associated with each processor is a *modified line table*, all of which are identical for a given column. This table is used to store addresses for all modified lines residing in caches in that column. With this information it is possible to route a row bus request either onto a specified column where the modified line resides or to memory. Second, a single tag bit is associated with each line in main memory indicating whether the contents are valid or invalid, that is, modified. This bit is necessary to prevent a request from acquiring stale data from memory while the modified line tables are in an inconsistent state.

A cache controller initiates four types of transactions: (1) a READ transaction, indicating that the processor wishes to read from a line that is not present in its cache, (2) a READ-MOD transaction, indicating that the processor wishes to modify a line that is not in modified mode in its cache, (3) an ALLOCATE transaction, indicating that the processor wishes to modify an entire line without regard to its prior contents, and (4) a WRITE-BACK transaction, indicating that main memory should be made current and the line changed to global state unmodified. A transaction requires multiple row and column bus operations to acquire and return the desired line. We will discuss each of these transactions in some detail to demonstrate the main features of the cache consistency protocol, which is presented formally in Appendix A.

[3]There are transition periods where the global state is indeterminant.

READ Transaction

A READ request is issued by a controller (*e.g.* controller A in figure 1) on its row bus. Some controller in the row always accepts the request, thereby accepting responsibility for assuring that the request is satisfied. If the requested line is in global state modified, then one controller in the row will recognize this fact (after looking in its modified line table) and will accept the request, responding after a fixed delay. This controller (C) now requests its column bus and transmits the request, which is picked up by the controller holding the modified data (D). A side effect of this column bus operation is that all the controllers on the column delete the line address from their modified line table. Controller D fetches the data from its cache, changing its mode from modified to shared, requests access to its column bus, and transmits the data. Controller C accepts the data from the column bus and transmits it on its row bus, where it is picked up by the requesting processor A and by the home column controller B. Controller B writes the line back to memory (M1), changing it to the global state unmodified (*i.e.*, setting the valid bit in memory).

If the requested line exists in global state unmodified, then the controller (B) on the home column of the line accepts the request. If it has the line in its cache, it requests the row bus and sends the data. If not, it requests its column bus and sends the request to memory (M1). When memory responds by sending the requested data, the controller B copies the data (possibly inserting the line into its own cache) and transmits it over the row bus to the requesting controller.

READ-MOD Transaction

A READ-MOD transaction is handled somewhat differently, beginning with the third bus operation. If the line is in global state modified, the controller holding the modified data (controller D in the example above) invalidates its copy of the line, and transmits the line on its *row* bus, where it is picked up by the controller (E) in the same column as the original requester (A). Controller E forwards the data over its column bus to A. A side effect of the column bus data transfer is that all the controllers in the column add the line address to their modified line table. Controller A enters the line into its cache in modified mode. Note also that main memory is not updated.

If the requested line is in global state unmodified, it is necessary to ensure that copies of the line, which may exist in any of the caches in the system, are purged. This is accomplished in four steps. First, the READ-MOD request is accepted by the controller (B) on the home column. Second, this controller forwards the request to main memory (M1). Third, memory transmits a *line invalidate* request along with a copy of the data (which it then marks invalid) on its column. Fourth, each controller in the column requests its row bus and transmits the line invalidate command, after which each controller in the system (other than the originator) checks for the presence of the line in its cache, purging it if found. The controller (B) on the same row as the originator sends the requested data along with the line invalidate request. A final bus operation is required (by controller A) to insert an entry into each modified line table in its column.

In case of a race between two requests for the same cache line (where at least one of the requests is a READ-MOD), the first request appearing on the home column (in the case of an unmodified line) or on the modified column (in the case of a modified line) determines the winner. The losing request is retransmitted (by the controller that previously accepted the request) on the row bus, where it is treated exactly as if it were a new request (but destined for the original requester). The reader is referred to the Appendix for details.

ALLOCATE Transaction

An ALLOCATE request is issued by a controller (*e.g.* controller A in figure 1) on its row bus when it receives a processor hint of an intention to modify the entire line, implying that the controller might wish to acquire local control of the line in modified mode. The ALLOCATE request is identical to the READ-MOD request, except that an acknowledge, rather than data, is returned to the requester. Thus, ALLOCATE is an optimization of READ-MOD that reduces bus traffic.

The allocate hint must be used carefully, because if there is any possibility that two processors might try to write parts of the same line simultaneously, the modifications of one may be lost. It is intended specifically for cases where entire blocks are to be written. It may be implemented in a manner that allows the processor to write a line before receiving the acknowledge of the ALLOCATE. Much of the benefit can be obtained by its inclusion in a few places, such as in I/O handlers, loaders, and memory allocators.

The cost of implementation is some additional complexity and an additional cache line state which signifies that the line can be written locally, but that the modified line table has not been updated. The allocate hint can be ignored entirely by the controller, since a succeeding write will then result in a READ-MOD transaction. Because of this and the fact that it is a minor variation of the READ-MOD transaction, it is not included in the formal protocol.

WRITE-BACK Transaction

A WRITE-BACK transaction is initiated by placing an operation on the column bus to remove the entry for that line from the modified line table. The controller may then proceed to write the line to memory. The table entry is removed first in order to avoid the problem where an outstanding request attempts to acquire the line, only to discover that it has already been written to memory.

Timing Considerations

The race occurring between two requests, at least one of which is a READ-MOD (or ALLOCATE), requires careful attention to ensure correct behavior. Notice that, in any given situation, exactly one column bus must be accessed to acquire the line exclusively. Thus, the normal bus arbitration mechanism may be used, just as in a single-bus multi, to resolve the race condition.

The valid bit in memory provides a robustness in the protocol that can greatly simplify the controller design. The only time that a controller is required to respond in a fixed time is when it must indicate on its row bus that the requested line resides in modified mode in its column, thus designating that column to be used instead of the home column. However, if the controller fails to respond under such a circumstance, the request is routed (incorrectly) onto the home column. As in the case of the losing home column READ-MOD request, this request will also be transmitted to main memory on the home column and retransmitted by main memory, since the line in memory is invalid. It is then forwarded onto the row bus of the originator, just as if it were an original request.

This robustness means that a controller can, on occasion, simply discard such requests without breaking the protocol. We expect that this feature will greatly simplify certain aspects of the controller design.

4. Synchronization

In a shared-memory multiprocessor where multiple processes are cooperating closely, it is imperative that efficient synchronization mechanisms be provided, *i.e.*, the operating system must not be involved routinely, and bus bandwidth must not be squandered. The techniques appropriate for a single-bus multi involve operations such as Test-and-Test-and-Set [RuSe84] which translate to multiple broadcast operations in the multiple bus multi, or involve a separate interface [BeKT87], which may not scale to hundreds or thousands of processors. Thus, better mechanisms are necessary. We have tentatively identified one such mechanism, which supports efficient implementations of test-and-set and can be readily extended to include combining operations [Rudo82]. The primitive is a remote test-and-set operation, which is executed wherever the modified line resides, or in memory if unmodified. It is implemented as an additional bus transaction, a variant of READ-MOD.

The test-and-set transaction returns a succeed/fail value to the processor. On success, the line addressed by

the test-and-set is moved to the cache of the successful processor. On failure, only the notification of failure is returned -- the line remains in the remote cache.

This mechanism works well for many locks, where contention is rare. In situations where test-and-set transactions would frequently fail -- causing excessive bus traffic -- it is desirable to queue up requests for mutually exclusive access to the same line. Such contention might arise, for example, with single-writer/single-reader synchronization, work queues, and barrier synchronization. A blocking operation is desirable, but such a primative introduces the problem of queuing requests and the necessity of handling queue overflow. However, cache *in*consistency can be exploited by implementing a distributed queue with a linked list, occupying a single word in different copies of the line. A node joins the tail of the queue by allocating space in its local cache for the line (marked *reserved*), clearing the designated word in its copy of the line, and initiating a SYNC transaction. The node with the copy at the end of the queue (or the modified copy, if there is no queue) receives the request and enters the id of the requesting node into the designated word of the line.[4] The entry in the modified line table is moved to the column of the new tail of the queue, so that a subsequent request will be routed to the correct node. When the node at the head of the queue is notified by its processor that it should release the line, it checks its copy to see if there is anything in the queue. If so, it forwards the line to the requesting node and deletes its copy. If not, it does nothing.

The actual operations on the line must also include a standard test-and-set operation to guarantee correctness. This will normally be very efficient, however, since a line that has been reserved locally with the SYNC transaction (but is not yet writable) will be recognized when a test-and-set operation is initiated, and the test-and-set will fail without requiring a bus operation. Whenever anything goes wrong -- for example, if a node is forced to purge a line to make room for a new one, or a process inadvertently writes in a line it shouldn't, breaking the locking protocol -- the scheme quickly degenerates to remote test-and-set, which guarantees correctness if not efficiency. Thus the sync primitive may be regarded as a cache hint, and the SYNC transaction can be left unimplemented. Unpredictable results may occur if a software locking protocol is not implemented correctly, but the underlying consistency protocol will guarantee that non-synchronizing operations will execute correctly.

This scheme provides an efficient mechanism for serial, mutually exclusive access to a line, collapsing bus traffic to a very low level. It has the nice property that it allows processors to spin-wait using test-and-set, yet (usually) provides first-come-first-served order of access. Thus processes can, for example, spin for a short time, then block if they wish. It also allows an unlimited number of queues to be implemented, since it requires no extra hardware or memory. Queue overflow will not occur because each node allocates space for any queue it joins. There are simple methods to avoid overflow even if two processes on the same node attempt to join the same queue, although strict FIFO order is no longer guaranteed.

A variation of the technique of exploiting the inconsistency of the caches can be used to implement barrier synchronization efficiently. This technique is currently being developed. Thus the entire synchronization extension to the protocol has not yet been included in the formal protocol.

Like some modern commercial multiprocessors, the Wisconsin Multicube does not guarantee complete serializability [FuKH87]. There are circumstances under which read operations and write operations may be observed to occur in a different order by different processors. Synchronization in such cases can only be ensured by the use of test-and-set. Thus, this operation must be carefully implemented to guarantee correct behavior.

5. Design Issues

There are many additional architectural and software issues to be addressed in the design of the Wisconsin Multicube. Among the most important are: (1) the selection of the processor and the design of the floating-point coprocessor and processor cache, (2) design of the snooping cache and controller, (3) design of the memory and I/O subsystems, (4) methods for reducing bus latency, (5) refinement of the cache consistency protocol, (6) analysis of packaging constraints, and (7) operating system and compiler issues. While a prototype system can be used to investigate software and synchronization issues, hardware design choices must be made before the prototype can be built. This requires the use of either simulation or analytical modeling.

Simulation has the advantage that it can be used to model a high degree of detail in an existing or proposed system. However, since very little data has been publish on the memory reference behavior of parallel programs, the simulation must be based on statistical distributions of references and reference types. In this case, the frequency and percentages of READ and READ-MOD requests on modified and unmodified data must be specified. While an analytical model requires the same information, it is often easier to implement, evaluate, and modify such a model. For these reasons, a mean-value performance model has been developed and used to measure the performance of the Wisconsin Multicube and to explore various design issues. The results of these studies and an explanation of the model can be found elsewhere [LeVe88]. Some of the results are reproduced here to demonstrate the impact of several important design choices. Continued analytical modeling and simulation are planned.

[4] If the line is globally unmodified, the request degenerates to a normal READ-MOD.

Preliminary Performance Analysis

The snooping cache is comparable in size to the main memory of many contemporary machines. Thus, one would expect bus traffic due to cache misses on private data to be comparable to page traffic on a conventional processor.[5] This should limit bus traffic essentially to shared data and I/O. All of the results in this section attempt to show the effect of bus request rates on performance for various design choices.

Figure 2 shows the predicted performance of the architecture for various rates of bus requests and various numbers of processors. Requests are assumed to be non-overlapping. Since our goal is to support 1K processors at roughly ninety percent utilization, the modified line table must be large enough and shared data accesses must be infrequent enough to produce an average access rate of less than twenty-five requests per millisecond per processor. This corresponds to a snooping cache miss rate of 0.25 percent for a processor generating 10 million memory requests per second.

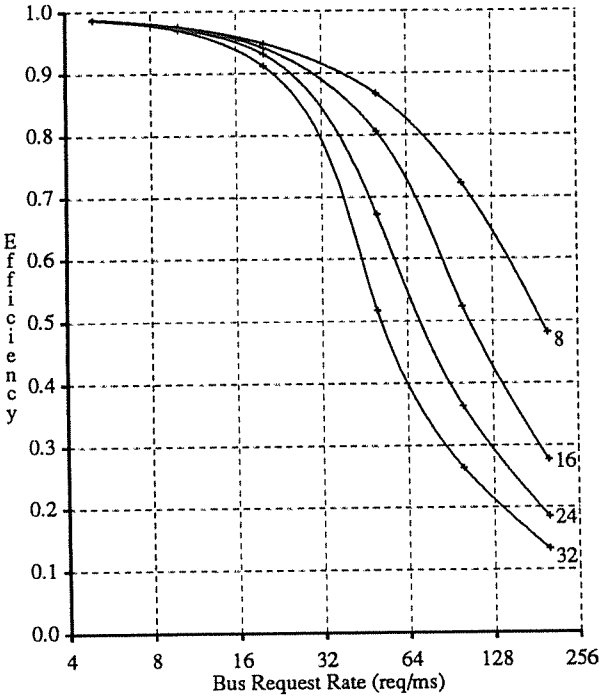

Figure 2: Efficiency versus Number of Processors per Row [LeVe88]. The numbers of processors per row are, from top to bottom, 8, 16, 24, and 32. The total number of processors is equal to the number per row squared. Efficiency is defined as the effective speedup compared to a system with no bus or main memory latency. A block size of 16 words is assumed. The probability that the requested data is in global state unmodified is 80 percent, and the probability that an invalidation operation is required for a write miss to unmodified data is 20 percent. The data is transferred at a rate of 1 bus word every 50 ns. The latency of both the snooping cache and main memory is 750 ns.

[5]The bus traffic may be lower because the line size will probably be smaller than typical page sizes.

Figure 3 shows that the effect of invalidations grows as bus request rates increase. The curves begin to converge as invalidations increase to the point where they saturate the available bus bandwidth. However, in the range of ninety percent processing power, the effect of increasing invalidations is very small. Thus, other factors are likely to dominate the level of performance.

Figure 4 shows the effect of increasing the block size on performance. However, block size and bus request rate are not independent. The two dashed lines represent the extreme cases where doubling the block size does not change the bus request rate (vertical line) and where doubling the block size halves the bus request rate (sloping line). Leutenegger and Vernon argue [LeVe88] that, for a more reasonable relationship between block size and bus request rate, a block size of 16 or 32 words would yield the best performance.

The degradation in performance for large blocks is primarily due to the increase in latency of each hop in the

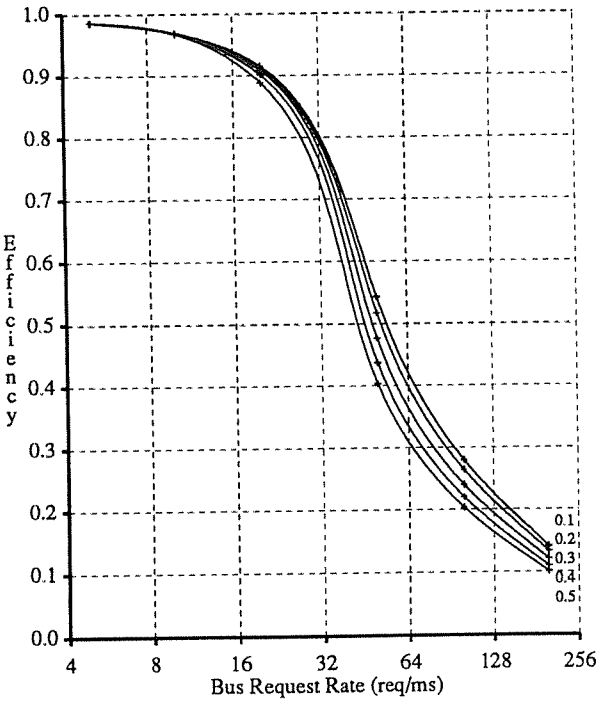

Figure 3: The Effect of Invalidations on Performance with 1K Processors [LeVe88]. The percent of write misses to shared data are, from top to bottom, 10, 20, 30, 40, and 50 percent. Other parameters are the same as those for Figure 2.

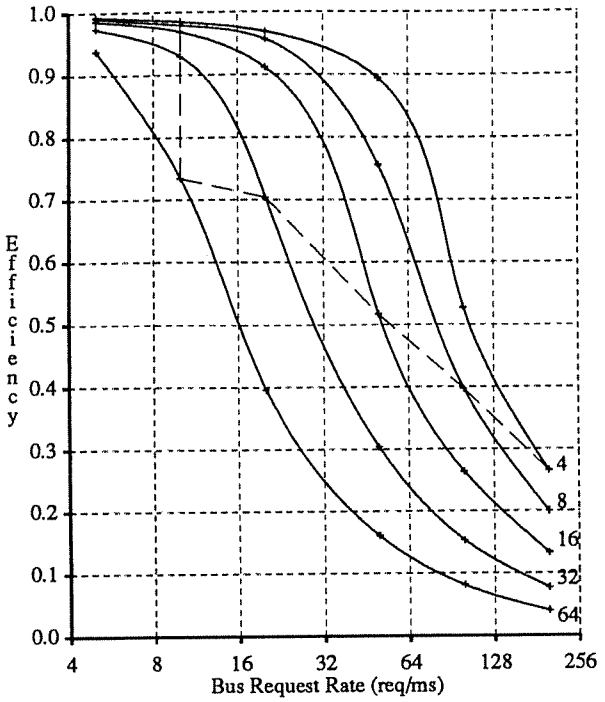

Figure 4: Effect of Block Size on Performance with 1K Processors [LeVe88]. The block sizes are, from top to bottom, 4, 8, 16, 32, and 64 bus words. Other parameters are the same as those for Figure 2.

path along which data is sent back to the requesting cache (the analysis assumes no overlapping of bus operations on sequential hops). A secondary effect is to increase the latency that competing requests experience due to the relatively long time for which a bus is held while the data is transferred. However, an effect which is not shown in figure 3 is that a large block size will reduce total waiting time for sequential, non-overlapped memory requests. Also, a larger block size will significantly lower invalidation traffic, since many operations (particularly I/O) will require one invalidation per block of data being referenced.

Techniques for Reducing Bus Latency

The selection of a line, or block, size actually involves two choices: the *transfer block* size and the *coherency block* size. The transfer block is the minimum amount of data transferred through the buses as a result of a memory request. The coherency block is the amount of data over which a single consistency check is performed. While coherency blocks must have a fixed, system-wide size, transfer blocks can be of variable size if, for example, the request specifies how much data should be returned. The only restriction seems to be that the transfer block size can be no larger than the coherency block size, in order to avoid multiple coherency checks for a single bus transaction [Good87].

The preliminary performance studies showed that large transfer blocks create long bus latencies and limit effective processing power, while large coherency blocks reduce invalidations. Thus it may be desirable to choose a transfer block, at least for certain transactions, that is much smaller than the coherency block.

The protocol as initially developed for the Wisconsin Multicube assumes that coherency and transfer blocks are the same size. Although the extensions are relatively straightforward, choosing blocks of different sizes results in substantial added complexity in the snooping protocol. Because the complexity of the snooping cache controller is of paramount importance, it is highly desirable to choose all transfer blocks and coherency blocks to be the same size.

A large coherency block size may result in inefficiencies due to *false sharing*[6]. However, a larger size will reduce the cost of invalidation operations, since an invalidation operation must generally be performed for each coherency block containing the structure being modified. Furthermore, the size of the modified line table is proportional to the number of coherency blocks, and therefore, inversely proportional to the coherency block size. Thus, the optimal coherency block size will be substantially larger than the small transfer block size needed to reduce bus latency. The problem then becomes one of achieving the low latency of small transfer blocks with the efficiency of large coherency blocks, but without the added complexity.

Data supplied in READ and READ-MOD transactions will traverse two buses (occasionally only one bus). Thus, the returned data will experience one full transfer block latency on the first leg, and an average of one half transfer block latency on the second leg. The first latency can be mostly eliminated by forwarding the data onto the second bus as soon as the first word arrives on the first bus. The second latency can be mostly eliminated by transmitting the requested word first. In fact, supplying words out of order is the only way of eliminating this delay (besides using small coherency blocks). These techniques introduce complexity at several levels. Thus their contributions to performance must be weighed carefully against their cost.

Forwarding requests has an additional disadvantage in that requests will still experience the queuing delays associated with long transfer blocks. An alternative scheme is to send the line in unequal parts. Supplying the requested word (and maybe a few more) in one operation and the rest of the line in a second operation will lead to the same problems. The use of two transfer block sizes may complicate the bus controller and the request may still experience long queuing delays. Another solution is to send the requested line in small fixed-size pieces. This

[6]False sharing occurs when two processors alternately read or write different parts of the same coherency block, resulting in the block's being moved repeatedly between the two processors as if the data were shared when in fact no sharing is occurring.

size can be optimized to reduce the bus latency while keeping bus header overhead to an acceptable level. Although the requesting processor may be delayed somewhat if it immediately accesses another word in the requested line, this effect should outweigh the bus latency caused by long transfer blocks.

Some of the latency reduction techniques have been analyzed elsewhere [LeVe88].

6. Architecture Scalability

The Wisconsin Multicube is a special case of a more general class of interconnection networks. The general *Multicube* consists of $N=n^k$ processors, where each processor is connected to k buses and each bus is connected to n processors. Figure 5 shows a Multicube for $n=4$ and $k=3$. A multi is a Multicube for which $k=1$. A hypercube is a Multicube for which $n=2$. The multiple broadcast capability proposed by Kumar and Raghavendra [KuRa87] superimposes a multicube on a mesh-connected computer (MCC). The Wisconsin Multicube is a two-dimensional Multicube ($k=2$). If 32 processors can be accommodated on a bus, the system can be extended to 1024 processors.

In the Wisconsin Multicube, main memory is divided up among the column buses. In general the placement of the memory is relatively unimportant, since very few of the bus requests require interaction with main memory. However, it is important that each line of memory have a *home bus* in order to assure sequentiality of access in cases of competing, mutually exclusive requests. The memory can be divided up on the bus, however, with each processor responsible for a piece of it, even incorporating it as part of its snooping cache that cannot be purged.

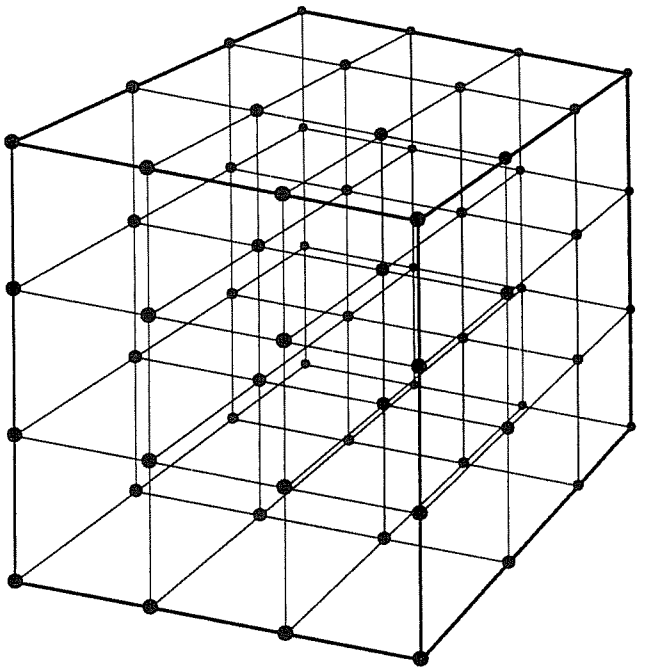

Figure 5: A 64-Processor/48-Bus Multicube with 3 Dimensions. Processors are represented as nodes. Buses are represented by straight lines.

The cache coherency protocol presented in Section 3 guarantees that, except under unusual circumstances, no more than four bus accesses are required for READs to unmodified lines (five if the requested line is modified). Likewise, READ-MODs to modified lines also require four bus accesses. However, in the case that a READ-MOD (or ALLOCATE) request is for an unmodified line, a broadcast operation is required. This includes $n+1$ row bus accesses and 3 column bus accesses. Fortunately, except for returning the data, all of these operations are very short, since they contain only an address and command information.

Given appropriate structures to locate modified data, the protocol can be extended. In a k-dimensional system, the number of buses that must be monitored by each snooping cache is k. The total number of buses is kn^{k-1}. Thus the bandwidth per processor grows as $\frac{kn^{k-1}}{N} = \frac{k}{n}$, and for fixed n, the bandwidth grows in proportion to k, precisely the rate at which the normal path length grows. Thus the architecture scales to provide the bandwidth needed for the common operations.

Several features of the Wisconsin Multicube, however, do not scale with k. The size of the modified line table must be large enough to recognize all modified lines of all the processors in a column.[7] The obvious extension requires the modified line table to recognize all modified lines in $\frac{N}{n}$ processors, although techniques have been identified for reducing the size with small increases in bus traffic under most circumstances. Invalidation operations scale less favorably, requiring approximately $\frac{N-1}{n-1}$ bus operations. Furthermore, the buses in different dimensions are not likely to be of the same speed.[8] While these factors tend to decrease the performance of a multidimensional system, there are other factors which have the opposite effect. READ requests to unmodified data are likely to be satisfied by some cache along the path to memory. Also, synchronization primitives, discussed in Section 4, allow for efficient sharing. Thus, these factors may be balanced in a multidimensional Multicube architecture to achieve scalable performance. This topic is a subject for future research.

[7] If the table is not large enough, modified lines will, on occasion, have to be written to main memory and changed to global state unmodified. This is why the modified line table is likely to be implemented as a cache.

[8] This packaging constraint can be overcome by adjusting the number of processors (and buses) in different dimensions so that the bus bandwidth per processor is constant in each dimension.

7. Summary

We have presented a large-scale, very-high-performance, shared-memory multiprocessor architecture that employs a snooping cache protocol over a grid of buses. The system presents the simplest known view of memory to the programmer: that of a single, coherent shared memory with no notion of geographical locality. Preliminary analysis suggests that this is a robust architecture, capable of good performance over a wide range of applications. Design and analysis of the Wisconsin Multicube are continuing.

We have identified a class of interconnection topologies -- the *Multicube*, of which the Wisconsin Multicube is a two-dimensional example. The topology scales to a very large number of processors, extending the simple programming model presented by the multi. Software developed for multis should be easily modified to run on a Multicube.

8. Acknowledgements

This work was supported in part by the National Science foundation, under grant DCR-8604224. We wish to acknowledge our colleagues Mary Vernon, Mark Hill, and Andrew Pleszkun for extensive discussion, comments and advice about the architecture and the paper. Scott Leutenegger and Mary Vernon have contributed much to our understanding of the architecture through their modeling efforts and the questions they forced us to address. Rae McLellan offered many key suggestions during the very early design. Dan Frank and Men-chow Chiang have made substantial contributions early in the Wisconsin Multicube project. Dave James pointed out the importance of being able to discard certain bus operations occasionally.

References

[BaWa87] Baer, J. L., and W. H. Wang, "Architectural Choices for Multilevel Cache Hierarchies," *Proc. of the 1987 Int. Conf. on Parallel Processing*, August 1987, pp 258-261.

[BeKT87] Beck, B, B. Kasten, and S. Thakkar, "VLSI assist for a multiprocessor," *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, October 1987, pp 10-20.

[Bell85] Bell, C. G., "Multis: a New Class of Multiprocessor Computers," *Science*, Vol. 228, April 26 1985, pp. 462-467.

[Butt85] *Butterfly Parallel Processor Overview*, BBN Laboratories, Inc., 1985.

[FuKH87] Fu, J., J. B. Keller, and K. J. Haduch, "Aspects of the VAX 8800 C Box design," *DEC Technical Journal*, Number 4, February 1984, pp. 41-51.

[GKLS83] Gajski, D., D. Kuck, D. Lawrie, and A. Sameh, "CEDAR -- a large scale multiprocessor," *Proc. of the 1983 Int. Conf. on Parallel Processing*, August 1983, pp. 524-529.

[Good87] Goodman, J. R., "Coherency for Multiprocessor Virtual Address Caches," *Proc. 2nd Int. Conf. on Architectural Support for Programming Languages and Operating Systems (ASPLOS-II)*, October 1987, pp 72-81.

[Good83] Goodman, J. R., "Using Cache Memory to Reduce Processor/Memory Traffic," *Proc. 10th Int. Symp. on Computer Architecture*, June 1983, pp. 124-131.

[GGKM83] Gottlieb, A., R. Grishman, C. P. Kruskal, K. M. McAuliffe, L. Rudolph, and M. Snir, "The NYU Ultracomputer -- designing an MIMD shared memory parallel computer," *The 5th ACM SIGACT/SIGOPS Symp. on Principles of Distributed Computing*, August 1986.

[KuRa87] Kumar, V. K., and C. S. Raghavendra, "Array processor with multiple broadcasting," *Journal of Parallel and Distributed Computing*, Vol. 4, (1987), pp. 173-190.

[LeVe88] Leutenegger, S., and M. K. Vernon, "A Mean-Value Performance Analysis of a New Multiprocessor Architecture," *ACM SIGMETRICS Conference on Measurement and Modeling of Computer Systems*, May 1988.

[Lund87] S. F. Lundstrom, "Applications Considerations in the System Design of Highly Concurrent Multiprocessors," *IEEE Trans. on Computers*, Vol. C-36, No. 11, November 1987, pp. 1292-1309.

[PBGH85] Pfister, G. F., W. C. Brantley, D. A. George, S. L. Harvey, W. J. Kleinfelder, K. P. McAuliffe, E. A. Melton, V. A. Norton, J. Weise, "The IBM Research Parallel Processor Prototype (RP3): Introduction" and Architecture," *Int. Conf. on Parallel Processing*, August 1985.

[Rudo82] Rudolph, L., "Software Structures for Ultra-parallel Computing," Ph.D. Thesis, Courant Institute, NYU, 1982.

[RuSe84] Rudolph, L., and A. Segall, "Dynamic decentralized cache schemes for MIMD parallel processors," *Eleventh Int. Symp. on Computer Arch.*, June 1984, pp. 340-347.

Appendix A

Formal Description of the Cache Consistency Protocol

There are four types of transactions -- READ, READMOD, ALLOCATE, and WRITEBACK -- which are initiated by the cache controller. Each transaction consists of a number of row and column bus operations necessary to satisfy the request. A bus operation contains up to four fields: a type, an originating node id (for routing replies), a line address, and possibly the contents of the line.

The formal protocol consists of a set of *procedures*, each of which directly corresponds to a specific bus operation. A transaction type is used as the procedure name, which is followed by a list of parameters specifying what specific actions are to be taken. Instead of one large procedure for each transaction type, each unique combination of transaction type and operations used in the protocol is defined separately. On a bus operation, all nodes on the bus, including the originator of the operation, execute the appropriate procedure.[9] The * symbol is placed in front of lines if (and only if) the memory unit is to execute that line.

It should be noted that the protocol presented here is *memoryless*. That is, once a node responds to a bus operation and performs the appropriate local actions (which may include queuing another bus operation) it can forget about the bus operation which it just processed. No extra state information needs to be kept. The only exception is for outstanding processor requests issued locally, some of which require information to be kept until the transaction is completed. Thus, the memoryless property is instrumental in reducing the complexity of the protocol.

The protocol assumes that, for all queues, operations are handled in a strict first-in, first-out (FIFO) order. This removes the problem of comparing or updating queue entries when operations such as data cache purges are performed. These properties allow hardware queues to be implemented efficiently and further reduces the complexity of the protocol.

The protocol also assumes the existence of a special row bus line called the *modified line*. It is used for requests to signify that some node has determined (by checking its modified line table) that the desired line resides in mode modified in a cache on its column. This signal is a logical OR of all of the nodes on the row, and is supplied (by at most one node) a fixed number of bus cycles after the request is placed on the bus.

Terminology used in the protocol is briefly described below. Procedures for the ALLOCATE transaction are not included since they contains only minor variations with respect to those used for READMOD.

[9]Many opportunities for optimization result from cases in which a node may be sending to itself, or can supply data to two nodes (one of which may be memory) at once. These have been included in the protocol, although they may be excluded from the implementation for simplicity.

Terminology

Transactions	
READ	result of a read miss
READMOD	result of a write miss
WRITEBACK	result of a data cache replacement of a modified line
Bus Operation Parameters	
ROW	row bus operation
COLUMN	column bus operation
REQUEST	request for a line
REPLY	reply containing the line or an acknowledge
INSERT	insert entry into modified line table
REMOVE	remove entry from modified line table
UPDATE	bus operation requiring a memory update
PURGE	bus operation requiring a line purge
NOPURGE	(indicates that no purge is needed on a column bus reply to a READ request)
MEMORY	bus operation destined for memory
Test Conditions	
id match	check if this node is transaction originator
row match	check if on same row as originator
column match	check if on same column as originator
line is 'mode'	check line mode (valid, invalid, shared, modified)
no modified signal	check if modified signal was supplied
on home column	check if this is the home column
table entry found	check for entry in modified line table
table overflow	check for modified line table overflow
overflow line in cache	check if table overflow line in data cache
remove failed	check if table remove operation succeeded
Local Operations	
write cache line	write line into data cache
write memory line	write line into memory
mark line 'mode'	mark mode of line in memory or a data cache (valid, invalid, shared, modified)
supply modified signal	indicate that the entry was found in the modified line table
select victim line	select data cache line for replacement
remove table entry	remove entry from modified line table
insert table entry	insert entry into modified line table
wait for continue	wait for data cache line replacement (WRITEBACK transaction) to complete
continue request	signal to continue processor request

READ Transaction Protocol

/* Initiate a READ transaction with a row bus request; first reserve space in the data cache (if necessary) with a WRITEBACK transaction */

- READ

```
if (line is invalid) then
    select victim line
    if (victim line is modified) then
        WRITEBACK (COLUMN, REMOVE)
        wait for continue
    mark line invalid
    READ (ROW, REQUEST)
```

/* row bus READ request for data; the request is either forwarded to the column where it resides in global state modified or to the home column */

- READ (ROW, REQUEST)

```
if (table entry found) then
    supply modified signal
    READ (COLUMN, REQUEST, REMOVE)
else if (on home column) and (no modified signal) then
    if (line is shared) then
        READ (ROW, REPLY)
    else
        READ (COLUMN, REQUEST, MEMORY)
```

/* column bus READ request for modified data; removing the modified line table entry guarantees access to the data; losing requests are reissued */

- READ (COLUMN, REQUEST, REMOVE)

```
remove table entry
if (remove failed) then
    if (row match) then
        READ (ROW, REQUEST)
else if (line is modified) then
    if (on home column) then
        READ (COLUMN, REPLY, UPDATE, MEMORY)
    else if (row match) then
        READ (ROW, REPLY, UPDATE)
    else
        READ (COLUMN, REPLY, UPDATE)
    mark line shared
```

/* column bus READ request for unmodified data; memory supplies the desired data if the line is valid, else it reissues the request */

- READ (COLUMN, REQUEST, MEMORY)

```
* if (line is invalid) then
*     READ (COLUMN, REQUEST, REMOVE)
* else
*     READ (COLUMN, REPLY, NOPURGE)
```

/* column bus reply to a READ request indicating that memory should be updated */

- READ (COLUMN, REPLY, UPDATE)

```
if (id match) then
    write cache line and mark line shared
    READ (ROW, UPDATE)
else if (row match) then
    READ (ROW, REPLY, UPDATE)
```

/* column bus reply to a READ request indicating that the memory on this column should be updated */

- READ (COLUMN, REPLY, UPDATE, MEMORY)

```
if (id match) then
    write cache line and mark line shared
else if (row match) then
    READ (ROW, REPLY)
* write memory line and mark line valid
```

/* column bus reply from memory to a READ request; no purge is required for a READ transaction */

- READ (COLUMN, REPLY, NOPURGE)

```
if (id match) then
    write cache line and mark line shared
else if (row match) then
    READ (ROW, REPLY)
```

/* row bus reply to a READ request */

- READ (ROW, REPLY)

```
if (id match) then
    write cache line and mark line shared
```

/* row bus reply to a READ request indicating that memory should be updated */

- READ (ROW, REPLY, UPDATE)

```
if (id match) then
    write cache line and mark line shared
if (on home column) then
    READ (COLUMN, UPDATE, MEMORY)
```

/* READ (ROW, UPDATE) and READ (COLUMN, UPDATE, MEMORY) are the same as WRITEBACK (ROW, UPDATE) and WRITEBACK (COLUMN, UPDATE, MEMORY) */

READMOD Transaction Protocol

/* Initiate a READMOD transaction with a row bus request; first reserve space in the data cache (if necessary) with a WRITEBACK transaction */

- READMOD

```
if (line is invalid) then
    select victim line
    if (victim line is modified) then
        WRITEBACK (COLUMN, REMOVE)
        wait for continue
    mark line invalid
    READMOD (ROW, REQUEST)
else if (line is shared) then
    READMOD (ROW, REQUEST)
```

/* row bus READMOD request for data; the request is either forwarded to the column where it resides in global state modified or to the home column */

- READMOD (ROW, REQUEST)

```
if (table entry found) then
    supply modified signal
    READMOD (COLUMN, REQUEST, REMOVE)
else if (on home column) and (no modified signal) then
    READMOD (COLUMN, REQUEST, MEMORY)
```

```
/* column bus READMOD request for modified data; removing the
modified line table entry guarantees access to the data; losing requests
are reissued */

• READMOD (COLUMN, REQUEST, REMOVE)

    remove table entry
    if (remove failed) then
        if (row match) then
            READMOD (ROW, REQUEST)
    else if (line is modified) then
        if (column match) then
            READMOD (COLUMN, REPLY, INSERT)
        else
            READMOD (ROW, REPLY)
        mark line invalid

/* column bus READMOD request for unmodified data; memory supplies the desired data if the line is valid, else it reissues the request */

• READMOD (COLUMN, REQUEST, MEMORY)

    * if (line is invalid) then
    *     READMOD (COLUMN, REQUEST, REMOVE)
    * else
    *     READMOD (COLUMN, REPLY, PURGE)
    *     mark line invalid

/* row bus reply to a READMOD request */

• READMOD (ROW, REPLY)

    if (id match) then
        write cache line and mark line modified
        READMOD (COLUMN, INSERT)
    else if (column match) then
        READMOD (COLUMN, REPLY, INSERT)

/* column bus reply from memory to a READMOD request; a purge of all copies of the line is required; the data cache on the home column must be purged first */

• READMOD (COLUMN, REPLY, PURGE)

    if (id match) then
        write cache line and mark line modified
        READMOD (COLUMN, INSERT)
        READMOD (ROW, PURGE)
    else
        mark line invalid
        if (row match) then
            READMOD (ROW, REPLY, PURGE)
        else
            READMOD (ROW, PURGE)

/* column bus reply to a READMOD request indicating that an entry should be inserted into the modified line table; on a table overflow an entry should be purged and the cache holding the modified line for that entry should write the line back to memory */

• READMOD (COLUMN, REPLY, INSERT)

    if (id match) then
        write cache line and mark line modified
    insert table entry
    if (table overflow) then
        if (overflow line is modified) then
            if (on home column) then
                WRITEBACK (COLUMN, UPDATE, MEMORY)
            else
                WRITEBACK (ROW, UPDATE)
            mark overflow line shared

/* row bus reply to a READMOD request also indicating that all shared copies of the line should be purged on the row; the home column data cache has already been purged */

• READMOD (ROW, REPLY, PURGE)

    if (id match) then
        write cache line and mark line modified
        READMOD (COLUMN, INSERT)
    else if (not on home column) then
        if (line is shared) then
            mark line invalid

/* row bus operation to purge all shared copies of a line; the home column data cache has already been purged */

• READMOD (ROW, PURGE)

    if (not on home column) then
        if (line is shared) then
            mark line invalid

/* insert an entry into the modified line table for this line; on a table overflow an entry should be purged and the cache holding the modified line for that entry should write the line back to memory */

• READMOD (COLUMN, INSERT)

    insert table entry
    if (table overflow) then
        if (overflow line is modified) then
            if (on home column) then
                WRITEBACK (COLUMN, UPDATE, MEMORY)
            else
                WRITEBACK (ROW, UPDATE)
            mark overflow line shared
```

WRITEBACK Transaction Protocol

```
/* Initiate a WRITEBACK transaction for a modified line by issuing a column bus REMOVE operation to delete the corresponding entry from all modified line tables in the column */

• WRITEBACK

    if (line is modified) then
        WRITEBACK (COLUMN, REMOVE)
        wait for continue
    mark line shared

/* write the line to memory; if the modified line table remove operation fails then some other bus operation will remove the data; in either case signal the processor request to continue */

• WRITEBACK (COLUMN, REMOVE)

    remove table entry
    if (id match) then
        if (remove succeeded) then
            if (on home column) then
                WRITEBACK (COLUMN, UPDATE, MEMORY)
            else
                WRITEBACK (ROW, UPDATE)
        continue request

/* forward the memory update request to the home column */

• WRITEBACK (ROW, UPDATE)

    if (on home column) then
        WRITEBACK (COLUMN, UPDATE, MEMORY)

/* write a line into memory */

• WRITEBACK (COLUMN, UPDATE, MEMORY)

    * write memory line and mark line valid
```