Scalable Shared-Memory Multiprocessor Architectures

New coherence schemes scale beyond single-bus-based, shared-memory architectures. This report describes three research efforts: one multiple-bus-based and two directory-based schemes.

Introduction

Shreekant Thakkar, Sequent Computer Systems
Michel Dubois, University of Southern California
Anthony T. Laundrie and Gurindar S. Sohi, University of Wisconsin-Madison

There are two forms of shared-memory multiprocessor architectures: bus-based systems such as Sequent's Symmetry, Encore's Multimax, SGI's Iris, and Stardent's Titan; and switching network-based systems such as BBN's Butterfly, IBM's RP3, and New York University's Ultra. Because of the efficiency and ease of the shared-memory programming model, these machines are more popular for parallel programming than distributed multiprocessors such as NCube or Intel's iPSC. They also excel in multiprogramming throughput-oriented environments. Although the number of processors on a single-bus-based shared-memory multiprocessor is limited by the bus bandwidth, large caches with efficient coherence and bus protocols allow scaling to a moderate number of processors (for example, 30 on Sequent's Symmetry).

Bus-based shared-memory systems use the bus as a broadcast medium to maintain coherency; all the processors "snoop" on the bus to maintain coherent information in the caches. The protocols require the data in other caches to be invalidated or updated on a write by a processor if multiple copies of the modified data exist. The bus provides free broadcast capability, but this feature also limits its bandwidth.

New coherence schemes that scale beyond single-bus-based, shared-memory architectures are being proposed now that the cost of high-performance interconnections is dropping. Current research efforts include directory-based schemes and multiple-bus-based schemes.

Directory-based schemes. Directory-based schemes can be classified as centralized or distributed. Both categories support local caches to improve processor performance and reduce traffic in the interconnection. The following "coherence properties" form the basis for most of these schemes:

- *Sharing readers.* Identical copies of a block of data may be present in several caches. These caches are called readers.
- *Exclusive owners.* Only one cache at a time may have permission to write to a block of data. This cache is called the owner.
- *Reader invalidates.* Before a cache can gain permission to write to a block (that is, become the owner), all readers must be notified to invalidate their copies.
- *Accounting.* For each block address, the identity of all readers is somehow stored in a memory-resident directory.

Presence flags. One cache-coherence scheme, proposed by Censier and Feautrier[1] in 1978, uses presence flags. In each

0018-9162/90/0600-0071$01.00 © 1990 IEEE

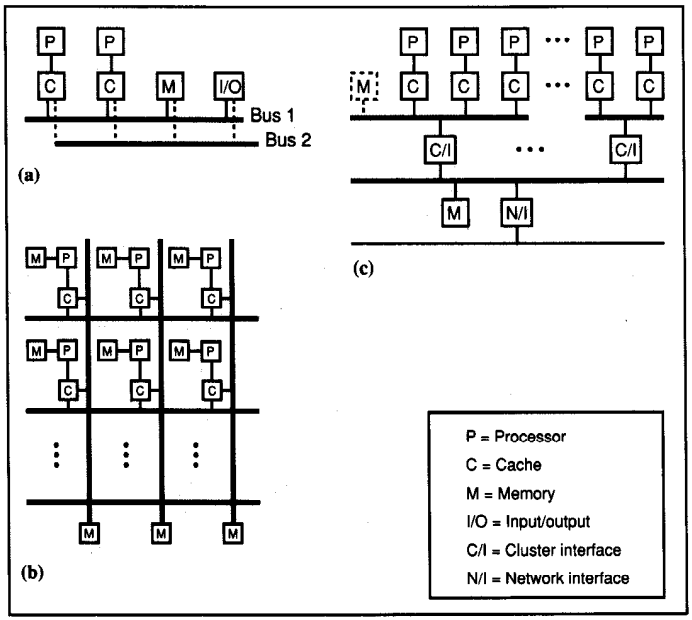

Figure 1. Extension of single-bus architectures to multiple-bus architectures: (a) one dimension; (b) two dimensions; (c) hierarchy.

memory module, every data block is appended with a single state bit followed by one presence flag per cache. Each presence flag is set whenever the corresponding cache reads the block. As a result, invalidation messages need only be sent to those caches whose presence bits are set.

In the presence-flag solution, unfortunately, the size of each memory tag grows linearly with the number of caches, making the scheme unscalable. The tag will be at least N bits, where N is the number of caches in the system. There may also be other bits stored in the directory to indicate line state.

Variations of this scheme use a broadcast mechanism to reduce the number of bits required in the directories. However, this introduces extra traffic in the interconnection and may degrade system performance.

In the central-directory scheme with presence flags, a cache miss is serviced by checking the directory to see if the block is dirty in another cache. When necessary, consistency is maintained by copying the dirty block back to the memory before supplying the data. The reply is thus serialized through the main memory. To ensure correct operation, the directory controller must lock the memory line until the write-back signal is received from the cache with the dirty block. Write misses generate additional invalidate messages for all caches that have clean copies of the data. Invalidate-acknowledgments must be received before a reply can be sent to the requesting cache. Note that the relevant line is locked while this is being done. Requests that arrive while a line is locked must be either buffered at the directory or bounced back to the source to be reissued at a later time. This may cause a loss in performance.

The performance of presence-flag schemes is limited by conflicts in accessing the main memory directory. The main memory and the tags can be distributed to improve the main memory's performance. However, the serialization of responses through the main memory and the locking of lines by the directory controller affect the performance of these cache-coherence schemes.

B pointers. Another alternative, being pursued by Agarwal et al.[2] and by Weber and Gupta,[3] requires each block to have a smaller array of B pointers instead of the large array of presence bits. Some studies of application programs[2-4] suggest that, because of the parallel programming model used, a low value for B (perhaps 1 or 2) might be sufficient. Since each shared data structure is protected by a synchronization variable (lock), only the lock — not the shared data structure — is heavily contested in medium- to large-grain parallel applications. Thus, the shared data only moves from one cache to another during computation, and only synchronization can cause invalidation in multiple caches. If B is small and the data is heavily shared, processors can thrash on the heavily shared data blocks. If B is large, the memory requirements are worse than for the presence-flag method.

Linked lists. Note that the presence-flag solution uses a low-level data structure (Boolean flags) to store the readers of a block, while the B-pointers method saves them in a higher level structure (a fixed array). Perhaps more-flexible data structures, such as linked lists, can be applied to the problem of cache coherence. Distributing the directory updates among multiple processors, rather than a central directory, could reduce memory bottlenecks in large multiprocessor systems.

A few groups have proposed cache-coherence protocols based on a linked list of caches. Adding a cache to (or removing it from) the shared list proceeds in a manner similar to software linked-list modification. Groups using this approach include the IEEE Scalable Coherent Interface (SCI) standard project, a group at the University of Wisconsin-Madison, and Digital Equipment Corporation in its work

with Stanford University's Knowledge Systems Laboratory. The SCI work, which is the most defined of the three, is covered in the report beginning on p. 74. Some features of the Stanford Distributed Directory (SDD) Protocol are outlined on pp. 78-80.

Bus-based schemes. Bus-based systems provide uniform memory access to all processors. This memory organization allows a simpler programming model, making it easier to develop new parallel applications or to move existing applications from a uniprocessor to a parallel system. Since the bus transfers all data within the system, it is the key to performance — and a potential bottleneck — in all bus-based systems.

Several architectural variations of bus-based systems have been proposed. Below, we describe two types — multiple-bus and hierarchical architectures. (See Figure 1.) One of these, the Aquarius multiple-bus multiprocessor architecture, is described in more detail in the report beginning on p. 80.

Multiple-bus systems. An obvious way to extend the single-bus system is to increase the buses. Studies by Hopper et al.[5] show that a system with multiple buses can provide higher bandwidth and performance than a system with wider buses. Multiple buses provide redundancy and extra bandwidth.

A simple extension, splitting a single bus into address and data buses, has a limited performance gain. The next choice is to duplicate buses. This scheme scales the bandwidth linearly with the memory. Synapse[6] had two buses, but they were used for redundancy rather than bandwidth (Figure 1a). (Arbitration and the coherence protocols become complex with multiple buses.)

The Wisconsin Multicube architecture[7] uses a grid of buses connecting the processors to memory. A processor resides at each crosspoint on the grid. The topology allows the system to scale to 1,024 processors. Each processor has a second-level cache that snoops on both the vertical and horizontal buses. (See Figure 1b.)

The Aquarius architecture is based on the same topology as the Wisconsin Multicube. However, it differs in the cache-coherence mechanism and in the distribution of memory modules. The system uses a combination of a snoopy cache-coherence protocol and a directory-based coherence protocol to provide coherency in the system. The memory is distributed per node, unlike the Wisconsin Multicube.

Hierarchical systems. In hierarchical systems, clusters of processors are connected by a bus or an interconnection network. (See Figure 1c.) In the three examples below, the intercluster connection is a bus, and the processors within a cluster are also connected via the bus. This is similar to a single-bus system.

Wilson[8] uses a simulation and an analytical model to examine the performance of a hierarchically connected multiple-bus design. The design explores a uniform memory architecture with global memory at the highest level. It uses hierarchical caches to reduce bus use at various levels and to expand cache-coherency techniques beyond those of a single-bus system. The performance study showed that degradation resulting from cache coherency is minimal for a large system.

The Diffusion Machine[9] is a scalable shared-memory system in which a hierarchy of buses and data controllers link an arbitrary number of processors, each having a large set-associative memory. Each data controller has a set-associative directory containing the state information for its data values. The controller supports remote accesses by snooping on the next bus in the hierarchy in both directions. A cache-coherence protocol enables data migration, duplication, and replacement.

The VMP-MC Multiprocessor[10] is another hierarchically connected multiple-bus multiprocessor system. The first-level cluster comprises processor-cache pairs connected by a bus and is similar to a single-bus system. However, instead of main memory, the bus has an interbus cache module that interfaces to the next bus in the hierarchy. This second-level bus has the main memory. Again, this system provides a uniform memory access. In this system, larger clusters are connected via a ring-based system to provide a large, distributed shared-memory system. ■

Acknowledgment

We would like to thank all the authors of the special reports that follow for their assistance and for their review of this introduction.

References

1. L.M. Censier and P. Feautrier, "A New Solution to Coherence Problems in Multicache Systems," *IEEE Trans. Computers*, Vol. C-27, No. 12, Dec. 1978, pp. 1,112-1,118.

2. A. Agarwal et al., "An Evaluation of Directory Schemes for Cache Coherence," *Proc. 15th Int'l Symp. Computer Architecture*, Computer Society Press, Los Alamitos, Calif., Order No. 861 (microfiche only), 1988, pp. 280-289.

3. W.D. Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Microprocessors," *Proc. ASPLOS III*, Computer Society Press, Los Alamitos, Calif., Order No. 1936, 1989, pp. 243-256.

4. S.J. Eggers and R.H. Katz, "A Characterization of Sharing in Parallel Programs and its Application to Coherence Protocol Evaluation," *Proc. 15th Int'l Symp. Computer Architecture*, Computer Society Press, Los Alamitos, Calif., Order No. 861 (microfiche only), 1988, pp. 373-382.

5. A. Hopper, A. Jones, and D. Lioupis, "Performance Evaluation of Widely Shared Multiprocessors," *Proc. Cache and Interconnect Workshop*, M. Dubois and S. Thakkar, eds., Kluwer Academic Publishers, Norwell, Mass., 1990.

6. S. Frank and A. Inselberg, "Synapse Tightly Coupled Multiprocessor: A New Approach to Solve Old Problems," *Proc. NCC 1984*, AFIPS, Reston, Va., 1984.

7. J.R. Goodman and P. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," *Proc. 15th Int'l Symp. Computer Architecture*, Computer Society Press, Los Alamitos, Calif., Order No. 861 (microfiche only), 1988, pp. 422-433.

8. A. Wilson, "Hierarchical Cache/Bus Architecture for Shared-Memory Multiprocessors," *Proc. 14th Int'l Symp. Computer Architecture*, Computer Society Press, Los Alamitos, Calif., Order No. 776 (microfiche only), 1987, pp. 422-433.

9. E. Hagersten, S. Haridi, and D.H.D. Warren, "The Cache-Coherence Protocol of the

Data Diffusion Machine," *Proc. Cache and Interconnect Workshop*, M. Dubois and S. Thakkar, eds., Kluwer Academic Publishers, Norwell, Mass., 1990.

10. H.A. Goosen and David R. Cheriton. "Predicting the Performance of Shared Multiprocessor Caches," *Proc. Cache and Interconnect Workshop*, M. Dubois and S. Thakkar, eds, Kluwer Academic Publishers, Norwell, Mass., 1990.

Shreekant Thakkar and **Michel Dubois** are the guest editors of this issue of Computer. Their photographs and biographies appear on p. 11.

Anthony T. Laundrie and **Gurindar S. Sohi** are authors of the report on the Scalable Coherent Interface. Their photographs and biographies appear on p. 77.

Distributed-Directory Scheme:

Scalable Coherent Interface

David V. James, Apple Computer
Anthony T. Laundrie, University of Wisconsin-Madison
Stein Gjessing, University of Oslo
Gurindar S. Sohi, University of Wisconsin-Madison

The Scalable Coherent Interface is a local or extended computer "backplane" interface being defined by an IEEE standard project (P1596). The interconnection is scalable, meaning that up to 64K processor, memory, or I/O nodes can effectively interface to a shared SCI interconnection.

The SCI committee set high-performance design goals of one gigabyte per second per node. As a result, bused backplanes have been replaced by unidirectional point-to-point links. One set of input signals and one set of output signals are defined. Packets are sent to the interconnection through the output link, and packets are returned to the node on the input link.

Although SCI only defines the interface between nodes and the external interconnection, the protocol is being validated on the least expensive and highest performance interconnection topologies, as illustrated in Figure 1.

To support arbitrary interconnections, the committee abandoned the concept of broadcast transactions or eavesdropping third parties. Broadcasts are "nearly impossible" to route efficiently, according to experienced switch designers, and are also hard to make reliable. Because of its large number of nodes and resulting high cumulative error rate, reliability and fault recovery are primary objectives of SCI. Therefore, its cache-coherence protocols are based on directed point-to-point transactions, initiated by a requester (typically the processor) and completed by a responder (typically a memory controller or another processor).

Sharing-list structures. The SCI coherence protocols are based on distributed directories. Each coherently cached block is entered into a list of processors sharing the block. Processors have the option to bypass the coherence protocols for locally cached data, as illustrated in Figure 2.

For every block address, the memory and cache entries have additional tag bits. Part of the memory tag identifies the first processor in the sharing list (called the head); part of each cache tag identifies the previous and following sharing-list entries. For a 64-byte cache block, the tags increase the size of memory and cache entries by four and seven percent, respectively, compared to the traditional eaves-

Project status and information sources

Scalable Coherent Interface. Simulation of the coherence protocols is now under way at the University of Oslo and Dolpin Server Technology in Oslo, Norway. The initial SCI simulation efforts focus on proving the specification's correctness rather than calibrating its performance.

Three University of Oslo researchers (Stein Gjessing, Ellen Munthe-Kaas, and Stein Krogdahl) are formally specifying the intent of the cache-coherence protocol and verifying that the cache updates prescribed by the SCI standard are specified correctly.

The University of Wisconsin's multicube group is now working with the SCI group.

IEEE's SCI-P1596 working group plans to freeze the base coherence protocols by this summer. The group will continue to explore optional coherence extensions to improve the performance of frequently occurring sharing-list updates. If you have interests in this area, please contact the SCI-P1596 working group chair: David B. Gustavson, Computation Research Group, Stanford Linear Accelerator Center, PO Box 4349, Bin 88, Stanford, CA 94309. Gustavson's phone number is (415) 926-2863, his fax number is (415) 961-3530 or 926-3329, and his e-mail address is dbg@slacvm.bitnet.

Stanford Distributed Directory. A group at Stanford University's Knowledge Systems Laboratory is working on simulations to determine the performance of their distributed-directory scheme using linked lists. Further information can be obtained from Manu Thapar, Knowledge Systems Laboratory, Department of Computer Science, Stanford University, 701 Welch Road, Palo Alto, CA 94304. Thapar's phone number is (415) 725-3849; his e-mail address is manu@ksl.stanford.edu.

Aquarius. The Aquarius group is evaluating the multi-multi architecture by simulation. Further information on that project can be obtained from Michael Carlton, University of California at Berkeley, Division of Computer Science, 571 Evans Hall, Berkeley, CA 94720. His phone number is (415) 642-8299, and his e-mail address is carlton@ernie.berkeley.edu.

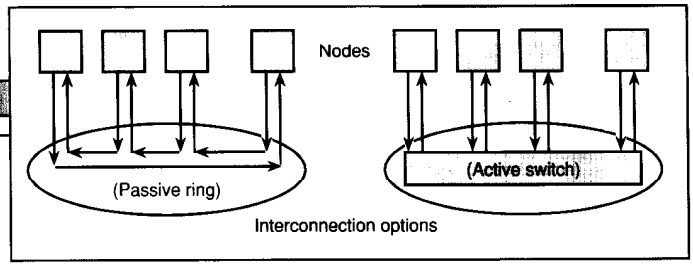

Figure 1. Scalable Coherent Interface (SCI) interconnection models.

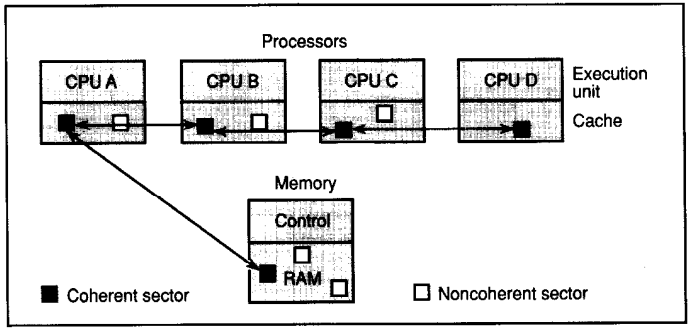

Figure 2. Distributed cache tags.

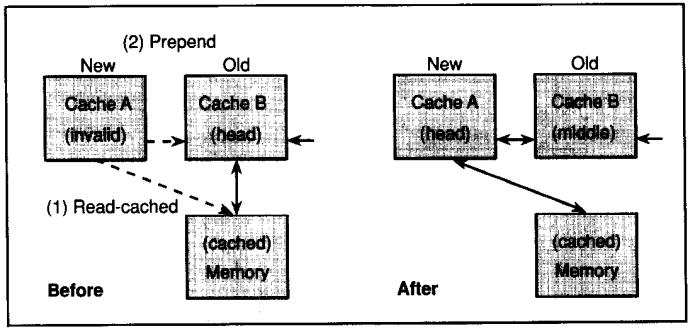

Figure 3. Sharing-list additions.

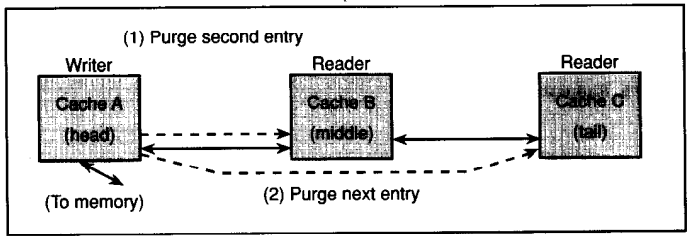

Figure 4. Head purging other entries.

dropping alternatives. However, snoopy protocols have other hidden costs; they require high-performance dual-ported cache-tag memories to allow execution of processor instructions while eavesdropping on other bus activity.

Sharing-list additions. Initially, memory is in the uncached state and cached copies are invalid. A read-cached transaction is directed from the processor to the memory controller. This changes the memory state from uncached to cached and returns the requested data. The data is returned and the requester's cache-entry state is changed from the invalid to the head state.

For subsequent accesses, the memory state is cached, and the head of the sharing list has the (possibly dirty) data. A new requester (Cache A) directs its read-cached transaction to memory, but receives a pointer to Cache B instead of the requested data. A second cache-to-cache transaction, called prepend, is directed from Cache A to Cache B. On receiving the request, Cache B sets its backward pointer to point to Cache A and returns the requested data, as illustrated in Figure 3.

The dotted arrow in Figure 3 illustrates a transaction directed between the processor (the requester) and memory or another processor (the responder). The solid line illustrates the sharing-list pointers. Note that memory cannot always forward the request directly to Cache B — that would create potential deadlocking dependencies.

Unlike the central-directory schemes, request transactions are never blocked at the memory controller; instead, all requests are immediately prepended to the head of the existing sharing list. Requests are added in FIFO order, as defined by the arrival of coherent requests at the memory controller.

Sharing-list removals. The head of the list has the authority to purge other entries to obtain an exclusive (and therefore modifiable) entry. The initial transaction to the second sharing-list entry purges that entry from the sharing list and returns its forward pointer. The forward pointer is used to purge the next (previously the third) sharing-list entry. The process continues until the tail entry is reached, as illustrated in Figure 4. As an option, the purges can be forwarded directly through the sharing-list entries.

Note that purge latencies increase linearly with the number of sharing readers.

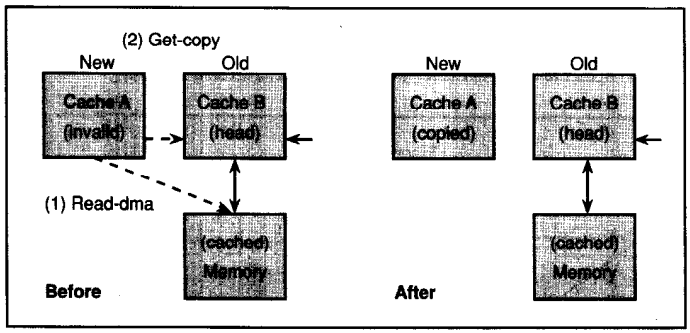

Figure 5. Optimized direct-memory-access reads.

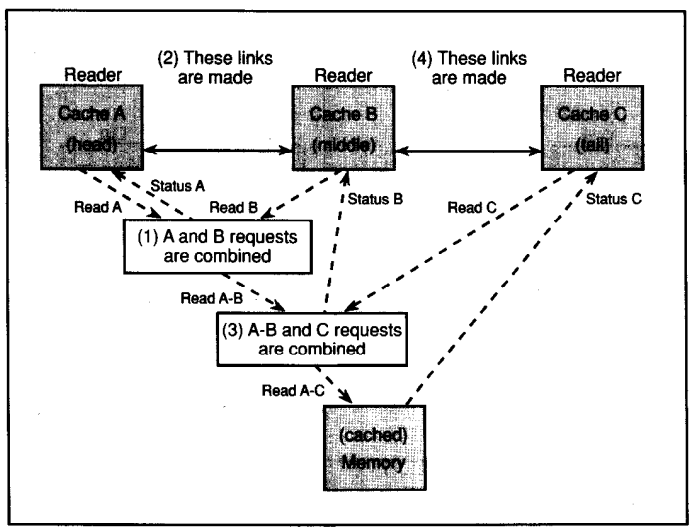

Figure 6. Request combining.

Since purge list sizes are often small, the linear latencies may be acceptable in many configurations.

Entries can also delete themselves from the list when they are needed to cache other block addresses. Since the linked list is distributed and doubly linked, multiple entries can delete themselves simultaneously. Special precedence rules are applied to avoid corruption of pointers when adjacent deletions are initiated simultaneously. To ensure forward progress, the entry closest to the tail has priority and is deleted first.

Standard optimizations. The basic coherence-protocol operations have been optimized to improve the performance of frequent events. We are considering other, more complex optimizations to improve the performance of large system configurations. These compatible extensions to the basic coherence protocols will be included as part of the SCI standard.

An optimized direct-memory-access controller generates read-check transactions to fetch its data from memory. If the addressed location is clean, the data is returned directly from memory; otherwise, the processor is redirected to the current sharing-list head. Thus, the DMA controller can fetch its data without joining the sharing list, as illustrated in Figure 5.

The frequent one-writer/one-reader (producer/consumer) form of data sharing is optimized. The invalidation of the writer (head) and the data fetches of the reader (tail) are both performed as direct cache-to-cache transactions between the head and tail of an established sharing list.

Request combining. One useful feature of linked-list coherence is the possibility of combining list-insertion requests in the interconnection to eliminate hot spots at or near heavily shared memory controllers. Such hot spots degrade performance not only of the requesting processor but also of other transactions that share portions of the congested connection path.

While queued in an active switch buffer, two requests to the same physical memory address (read A and read B) can be combined. The combining generates one response (status A), which is immediately returned to one of the requesters, and one modified request (read A-B), which is routed towards memory. Additional requests (read C) can also be combined with the modified request, as illustrated in Figure 6.

Read transactions and add transactions (add to previous value) can be combined in the interconnection or at the memory controller's front end. Coherent-request combining is simpler than noncoherent fetch-and-add combining,[1] since state need not be saved in the interconnection while the modified request is being forwarded to memory.

SCI's optional extensions. The latency of distributing data or purges to large numbers of readers currently scales linearly with the number of read-sharing processors. We are investigating the use of

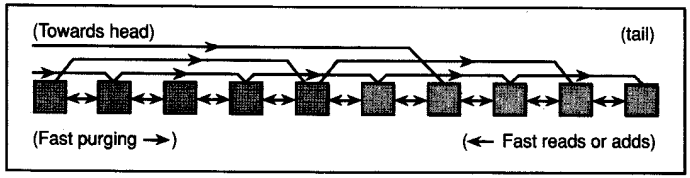

Figure 7. Redundant sharing-list pointers.

redundant pointers to reduce these linear delays to logarithmic latencies (order log(N), where N is the number of read-sharing processors). The redundant pointers can be created while the request combining is being performed, to provide the binary tree-like structure illustrated in Figure 7.

The redundant pointers could be used by multiple readers, to request early copies of heavily shared data, or by a writer, to quickly purge stale copies when a new data value is written.

Synchronization. In shared-memory architectures, locks are the primary form of synchronization for large-scale multiprocessors and must be handled efficiently. The SCI options include efficient synchronization primitives for large-scale multiprocessors. A queued-on-lock-bit idea, described by Goodman, Vernon, and Woest,[2] provides FIFO access to synchronization variables. Since linked cache entries form a queue, little additional hardware is needed to implement an SCI variant of this scheme. The advantage of the queued-lock scheme is that (except for replacements) lock requests are serviced in FIFO order and only $O(N)$ transactions are generated. ■

Acknowledgments

The IEEE P1596 Scalable Coherent Interface project was started as a study group, under the name of SuperBus, by Paul Sweazey. Dave Gustavson is now the chair and is responsible for the continuing development efforts. Others initially or currently involved with the cache-coherence issues include Knut Alnes, Jim Goodman, Marit Jensen, Ernst Kristiansen, Stein Krogdahl, John Moussouris, Ellen Munthe-Kaas, Alan Smith, and Hans Wiggers.

We would like to acknowledge recent contributions from Jim Goodman and others at the University of Wisconsin that have simplified the basic proposals and triggered many of the SCI project's continuing investigations.

References

1. G.F. Pfister et al., "The IBM Research Parallel Processor Prototype (RP3): Introduction and Architecture," *Proc. Int'l Conf. Parallel Processing*, Computer Society Press, Los Alamitos, Calif., Order No. 637 (microfiche only), 1985, pp. 764-771.

2. J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proc. ASPLOS III*, Computer Society Press, Los Alamitos, Calif., Order No. 1936, 1989, pp. 64-75.

David V. James, a research scientist at Apple Computer, is a major participant in several IEEE bus standards. He is the chair of the IEEE P1212 CSR Architecture working group as well as a member of the directly affected bus standards (IEEE P896.2-Futurebus+, P1596-SCI, and P1394-Serialbus). His research interest is scalable interconnection architectures, from low-cost mouse interfaces to high-performance massively parallel processors.

James holds BS and MS degrees in electrical engineering and a PhD degree in electrical engineering and computer science from the Massachusetts Institute of Technology. He is a member of IEEE, ACM, Eta Kappa Nu, and Tau Beta Pi.

Anthony T. Laundrie received the BSEE and MSEE degrees in 1987 and 1990 from the University of Wisconsin-Madison. The past four years have been spent writing design automation software and integrating memory hardware for the Astronautics ZS-1 minisupercomputer.

Now back in school at the University of Wisconsin-Madison, he is studying memory architectures for high-performance systems in greater detail. Laundrie is a member of IEEE, Eta Kappa Nu, Tau Beta Pi, and Phi Kappa Phi.

Stein Gjessing is an associate professor at the University of Oslo, Norway. He is presently head of the Department of Informatics. His research interests are in concurrent programming, operating systems, and formal specification and verification of computer programs. His work on SCI is supported by the Norwegian Research Council NTNF.

Gjessing received his PhD in computer science from the University of Oslo in 1985.

Gurindar S. Sohi has been with the Computer Sciences Department at the University of Wisconsin-Madison since September 1985. He is currently an assistant professor. His interests are in computer architecture, parallel and distributed processing, and fault-tolerant computing.

Sohi received his BE degree, with honors, in electrical engineering from the Birla Institute of Science and Technology, Pilani, India, in 1981 and the MS and PhD degrees in electrical engineering from the University of Illinois, Urbana-Champaign, in 1983 and 1985.

Distributed-Directory Scheme:

Stanford Distributed-Directory Protocol

Manu Thapar and Bruce Delagi,
Digital Equipment Corporation and Stanford University

The Stanford Distributed-Directory (SDD) coherence protocol is based on a singly linked list of distributed directories. Sharing-list additions and removals are handled differently than in the Scalable Coherent Interface (SCI) protocol. Figure 1 shows the operations used to add a cache to the list. A shaded arrow depicts a single message between two nodes. (Note the distinction from the dotted arrow used in the preceding SCI report to indicate a transaction involving two messages, a request and a reply.)

Reads. On a read miss, a new requester (Cache C) sends a read-miss message to memory as shown in Figure 1. The memory updates its head pointer to point to the requestor and sends a read-miss-forward signal to the old head (Cache B). On receiving the request, Cache B returns the requested data along with its address as a read-miss-reply. When the reply is received at Cache C, the data is copied and the pointer is made to point to Cache B. Read misses thus result in three messages instead of four as in the case of the SCI protocol. The three-message scheme is not always safe to use; when queues are full, the SCI four-message scheme must still be supported to avoid message-queue deadlocks.

Writes. For writes, write buffering along with weak ordering[1] lets the processor proceed immediately without stalling. A write is considered issued when the cache sends a write miss. A write is considered performed when the caches receives a write-miss-reply. On a write miss, a requestor (Cache D) sends a write-miss message to memory, as shown in Figure 2. The memory updates its head pointer to point to the requester and sends a write-miss-forward signal to the old head (Cache C). Cache C invalidates its copy and sends a write-miss-forward signal to the next cache in the list (Cache B). Cache C also sends the data to Cache D as a write-miss-reply-data signal. When Cache B receives the write-miss-forward signal, it invalidates its copy and sends a write-miss-forward signal to the next-cache (Cache A). When the write-miss-forward signal is received by the tail (Cache A) or by a cache that does not have a copy of the line — a case that may happen on replacement — that cache sends a write-miss-reply to the requestor. The write is considered performed when the requestor has received both the write-miss-reply-data and the write-miss-reply. Write misses result in about half the messages required for the base SCI protocol.

Write misses in caches that are part of the shared list are handled similarly and are described in detail in an earlier report.[2]

The latency of write misses may be a cause for concern, since caches linked in the list must be invalidated sequentially. However, if writes to a line occur frequently, the number of caches to be invalidated between writes will be small. Thus, cases when the latency is large will be infrequent. Additionally, write buffering[1] is used to reduce the effect of the sequential invalidation operations.

Pending signals. A cache line would be in the writing-or-reading state after generating a read miss or a write miss and before receiving a read-miss-reply or a write-miss-reply. If the cache line is in the writing-or-reading state and receives a read-miss-forward or a write-miss-forward signal, the forwarded signal is stored in the line's cache-pointer field. The state is changed to note that a forwarded signal has been stored. These stored signals, called pending signals, are serviced when the reply to the local read or write miss is received. If multiple transactions are pending for the same line, the caches form a distributed queue of pending signals. The requests are thus serviced in a pipelined manner that eliminates the directory contention of a centralized-directory protocol.

The naive forwarding of request signals can result in deadlocks. To avoid such deadlocks, forwarding does not occur under certain conditions; instead, a reply is generated as in the SCI protocol.

Replacement. Replacement of lines that are linked in a list is handled by invalidating the lower part of the list. A doubly linked list, as used by the SCI protocol, may be used to "patch" the list in case of replacements. However, in practice, performance improvement depends on the list lengths and access patterns. The improvement will probably be small, especially if replacement of shared writable data is infrequent. Some optimizations may be made to improve performance of the SDD protocol where long lists are expected. For example, in the case of read-only data, it's unnecessary to form a linked list of caches that contain shared copies. Replacement can be done directly, involving only a local operation. Page replacement of read-only data can be handled by software-based mechanisms.

Synchronization. A distributed-directory cache-coherence protocol allows efficient implementation of locks at minimal extra cost. Familiar microprocessor architectures have some form of atomic test-and-set instruction to implement spin locks.[3] The test-and-set instruction sets the value of a memory location and atomically returns the old value. When a process wants access to a lock, the processor performs the test-and-set instruction. If the operation is successful, the processor continues; otherwise, the processor repeatedly tries to access the lock until it is successful.

Spinning on a test-and-set instruction can cause a lot of network traffic each time a lock is released. A better alternative is a test-test-and-set sequence, where the first test is done in the cache and the test-and-set only if the first test is successful. This will reduce network traffic, but network traffic due to lock acquisition will still be $O(N^2)$, where N equals the number of pro-

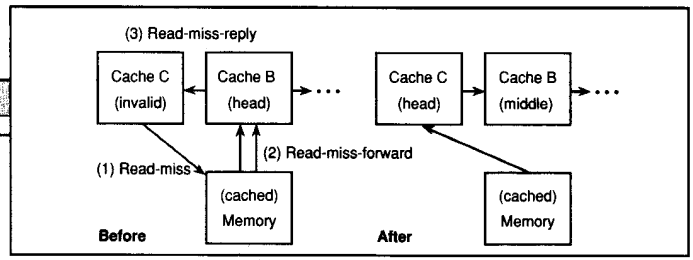

Figure 1. Sharing-list additions for the Stanford Distributed-Directory (SDD) protocol.

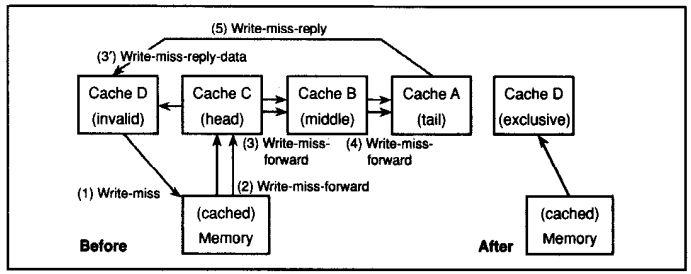

Figure 2. Sharing-list removals for the SDD protocol.

cessors contending for the lock. Further, such lock implementations can result in starvation, since the accessing of locks is not fair (perhaps due to differences in the network distance between a lock and its contenders).

The SDD protocol allows a lock implementation that minimizes network traffic. Lock requests are queued and normally serviced in FIFO order. The network traffic is only $O(N)$. Starvation is eliminated if there is one process per processor.

Fine-grain locking is provided by having a lock state per cache line.[4] The main advantage of such a scheme is that the data can be obtained at the same time as the lock.

The code for a lock procedure is

```
procedure lock(var)
begin
  Queue-Lock(var)
  while Test&Set(var)
    Queue-Lock(var)
end
```

A distributed queue of caches waiting for a lock is formed by a mechanism similar to the one used to form a distributed queue of pending signals. The implementation of locks requires a few extra states and signals.

When a queue-lock instruction is executed, a line in the lock-in-progress state is allocated, and a lock-miss signal is sent to the directory. If no other cache has locked the line, the directory sends a lock-granted signal along with the data. The first cache that receives the lock-granted signal is considered the upstream end. Otherwise, the directory updates its cache pointer to point to the requesting cache and forwards the lock-miss signal to the cache previously pointed to by the cache pointer. The cache receiving this forwarded signal stores the address of the requesting cache (the downstream pointer) in its cache-pointer field. A set-upstream-pointer signal with the old value of the cache pointer is also sent to the requesting cache. The last requesting cache is considered the downstream end. The directory points to the downstream end. The requesting cache stores this upstream pointer in its data field. In this way, a doubly linked list of caches waiting for a lock is formed.

Lines in the lock-in-progress state do not have valid data, so these lines can store the upstream pointers for the doubly linked list without requiring extra memory. The lock-in-progress state has a few sister states used to ensure receipt of the signals required to form the doubly linked list.

The test-and-set instruction returns a value of "true" until the lock is obtained at the local cache. A queue-lock instruction may be issued more than once. If the associated line has already been allocated in the cache, the instruction is redundant and has no effect. The queue-lock instruction in the while loop rejoins the queue of nodes waiting for a lock in case of replacements.

More than one processor may share a cache, so there may be multiple processes per processor contending for the same lock. In that case, each process executes the lock procedure.

The first queue-lock request to reach the cache causes a line in the cache to be linked to the doubly linked list of caches waiting for a lock. The processes associated with a cache contend for the lock by checking the cache locally, without generating any network traffic.

When a lock-granted signal is received, a process has to obtain and release the lock to allow other caches waiting for the lock to get it. If no other cache wants the lock, the line is held locally. Processes that share the cache can now lock and unlock the line very efficiently in the cache without requiring any network traffic.

After the lock has been used and released once locally, and if a lock-miss-forward signal is received, the lock is granted to the next downstream cache in the queue. This ensures fairness between processors. In the SDD protocol, the grant signal does not have to go through the directory, and the data is passed between the caches. The doubly linked list allows lock-grant signals to flow upstream. This is sometimes required for process migration.

The advantages of this scheme are that it (1) services lock requests, except for replacements, in FIFO order; (2) eliminates starvation, if there is one process per cache; and (3) requires only $O(N)$ operations. A similar scheme has been proposed by Goodman, Vernon, and Woest.[5] However, their original proposal required broadcast transactions and interactions with the main memory, which would have generated more network traffic.

For details on the SDD cache-coherence protocol, see our earlier report.[2] ■

Acknowledgments

We would like to thank Mike Flynn, Greg Byrd, and Max Hailperin for their comments and suggestions and members of the DEC High-Performance Systems Group and the Stanford Computer Systems and Knowledge Systems Labs for their support.

References

1. M. Dubois, C. Scheurich, and F. Briggs, "Memory Access Buffering in Multiprocessors," *Proc. 13th Int'l Symp. Computer Architecture*, Computer Society Press, Los Alamitos, Calif., Order No. 719 (microfiche only), 1986, pp. 434-442.

2. M. Thapar and B. Delagi. "Design and Implementation of a Distributed-Directory Cache-Coherence Protocol," Tech. Report 89-72, Knowledge Systems Laboratory, Stanford University, 1989.

3. T. Anderson, "The Performance of Spin-Lock Alternatives for Shared-Memory Multiprocessors," *IEEE Trans. Parallel and Distributed Systems*, Vol. 1, No. 1, Jan., 1990, pp. 6-16.

4. P. Bitar and A. Despain, "Multiprocessor Cache Synchronization Issues, Innovations, Evolution," *Proc. 13th Ann. Int'l Symp. Computer Architecture*, Computer Society Press, Los Alamitos, Calif., Order No. 719, 1986, pp. 424-433.

5. J.R. Goodman, M.K. Vernon, and P.J. Woest, "Efficient Synchronization Primitives for Large-Scale Cache-Coherent Multiprocessors," *Proc. ASPLOS III*, Computer Society Press, Los Alamitos, Calif., Order No. 1936, 1989, pp. 64-75.

Manu Thapar is a senior engineer in the Highly Parallel Systems and Applications Group at Digital Equipment Corporation. He is currently involved in the design of parallel systems. His interests include parallel architectures, memory management, VLSI, and system modeling.

Thapar received his BS in electrical engineering from Punjab University, India, in 1985 and his MS in electrical and computer engineering from Louisiana State University in 1987. He is currently working toward his PhD in electrical engineering at Stanford University. He is a member of Eta Kappa Nu and a student member of the IEEE Computer Society.

Bruce Delagi is an engineering group manager at Digital Equipment Corporation and a consulting professor of electrical engineering and computer science at Stanford University. He is now involved in the design of highly parallel systems. Long ago, he was one of the architects for the PDP-11 and was named as inventor on several related patents. He led the implementation effort for the PDP-11/45 and shortly after completion of this machine did time in engineering management (for the PDP-11 group) and corporate strategic planning (as manager of strategic planning for DEC engineering). He returned recently to technical contribution as co-director of the Advanced Architectures Project at Stanford's Knowledge Systems Laboratory.

Delagi received an SB from Massachusetts Institute of Technology in 1964 and an MSEE in 1966 from Purdue University.

Multiple-Bus Shared-Memory System:

Aquarius Project

Michael Carlton, University of California at Berkeley
Alvin Despain, University of Southern California

Multi-multi architecture. The Aquarius project at the University of California at Berkeley has been investigating multiprocessor systems composed of multiple buses, with the buses connected via different interconnection networks. The simplicity and efficiency of a single-bus multiprocessor[1] first led Goodman and Woest[2] to consider it as a building block for the Wisconsin Multicube project's large multiprocessors (that is, much larger than 30 processors). Goodman's work, in turn, led the Aquarius group to develop and investigate the architecture described here, which we call a multi-multi.

Figure 1 shows the multiple-bus system architecture. It has several architectural features in common with the Wisconsin Multicube project, but it differs noticeably in the methods used to provide cache coherency and in the distribution of memory modules. The example in Figure 1 contains only two dimensions, but the architecture is designed to handle several dimensions with a moderate number of processors per bus. It provides scaling to a large number of processors in a system. For example, a three-dimensional system with eight processors per bus would contain 512 processors.

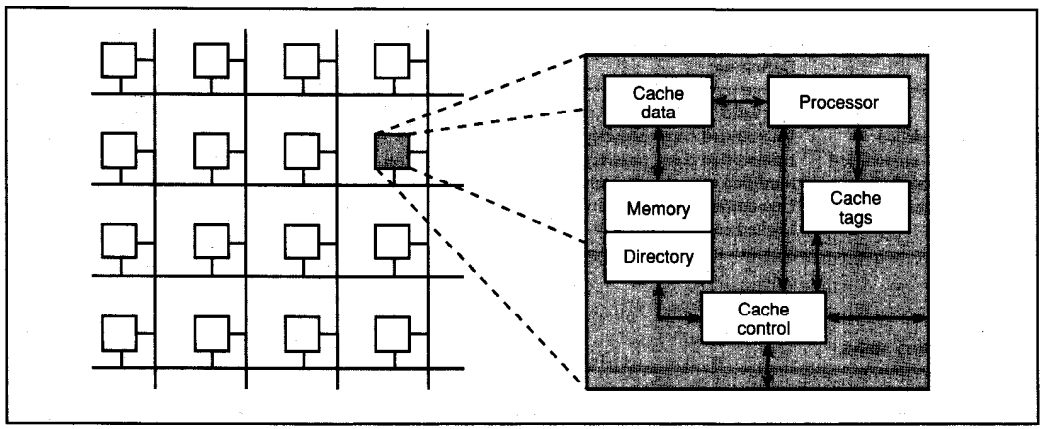

Figure 1. The multiple-bus system architecture.

A key characteristic of the architecture is the amount of bandwidth it provides. If the number of dimensions is d and the number of processors per bus is p, then the number of processors in the system is $n = p^d$. The number of buses (and hence total bandwidth) is dn/p, so the amount of bandwidth available per processor is d/p. This shows that as the dimensionality of the system increases, so does the amount of bandwidth available. Increasing the dimensionality also increases the maximum number of bus broadcasts required for a transaction to travel between processors. In the worst case, where every broadcast requires d steps, the effective bandwidth per processor is constant at $1/p$. Providing greater bandwidth as the number of processors increases supports the goal of designing a scalable system.

Another distinctive feature is that memory modules are assigned to nodes rather than attached to buses. All memory appears in the global shared-address space. The memory is interleaved on high bits so as to assign a large section of the address space to each of the physically local memories. This arrangement provides efficient support for private data and a natural division of directory information. Thus, each node can, if desired, run completely independently of all other nodes, or it can share to any desired degree. Sharing is most efficient among processors that share a bus but is possible among all processors in the system.

Nodes. Each node in the architecture contains a microprocessor, memory, and a cache. The processor is a high-performance microprocessor. The local memory is large, preferably at least four megabytes, and is tagged with directory information on a per-block basis. The caches are fairly large (128 kilobytes each for both the data and instruction parts). They consist of the data store and a multiple-ported tag store, with ports for the processor and each external bus the node attaches to.

Multi-multi protocol. The cache-coherence protocol for the multi-multi architecture combines features of snooping cache schemes, to provide consistency on individual buses, and features of directory schemes, to provide consistency between buses. The snooping cache component can take advantage of the low-latency communication possible on shared buses for efficiency, yet the complete protocol will support many more processors than a single bus can. The resulting protocol naturally extends cache coherence from a multi to a multi-multi.

The protocol uses split transactions for certain coherency actions. If a given coherency action cannot be completed in a single bus cycle (normally because insufficient information is available on that bus), it will be split, that is, some cache on the bus will relay the transaction to another bus. The cache that originated the transaction will enter a pending state until it receives a completion transaction.

The unit of coherence is a cache block, that is, the protocol handles cache blocks as indivisible items. Each node in the system is identified by the address range contained in its local memory. Each block is logically associated with the node whose memory contains the block. This node is called the root node for that block. The root node contains the directory information for the block and serves as a synchronization point for concurrent accesses.

Cache and directory states. Table 1 describes the basic cache states.

The shared states indicate that other caches in the system may have a copy of the same block. The local state guarantees that any other copies of the block will be on only the local bus. (See the section on local sharing.)

The exclusive unmodified state is used only for blocks at their root node to support efficient access to private data. The locked state, an extension of the cache-lock-state protocol described by Bitar and Despain,[3]

Table 1. Cache states.

Cache State	Definition
Invalid	Not valid
Shared-unmodified	Unmodified copy, possibly shared
Local-shared-unmodified	Unmodified copy, possibly shared on local bus only
Exclusive-modified	Modified copy, exclusively owned
Exclusive-unmodified	Unmodified copy, exclusively owned
Locked	Locked copy, exclusively owned

Table 2. Directory states.

Directory State	Definition
Uncached	No cached copies exist
Shared-unmodified	Unmodified, shared copies exist
Local-shared-unmodified	Unmodified, only locally shared copies exist
Exclusive-modified	A modified, exclusive copy exists
Exclusive-root	A copy (possibly modified) exists at the root only
Locked	A locked copy exists

supports synchronization primitives. A processor locks a block when it needs exclusive access to the block; a block in locked state cannot be accessed by other caches. The processor then unlocks the block when it is through modifying the block. The implementation of the cache also uses a few other states to indicate that the cache is arbitrating for the bus or that it has a read or write access pending after a transaction has been split.

The directory is distributed with the individual memories. The directory maintains information about the state of each block, as well as the location of a single bus on which the block resides. This information is used when the block is cached by one node for write access or is shared locally among nodes on a single bus; in these cases, bus transactions can be directed to only that bus.

Table 2 describes the basic directory states. The shared states allow for any number of cached copies, including zero. The exclusive root state does not indicate if the block has been written or not. This allows a root write to an exclusive-unmodified block to change state to exclusive-modified without updating the directory or requiring a bus transaction (either of which would require stalling the processor).

Concepts. The multi-multi protocol contains several important concepts that enable efficient performance.

Local sharing. The amount of sharing among nodes significantly affects system performance. Fortunately, the number of shared copies of a block tends to be low in practice.[4] This leads to the observation that if the blocks are shared among the nodes of a single bus, then the protocol can be optimized for this case. This is referred to as local sharing.

The multi-multi protocol takes advantage of local sharing by observing when a block is shared only locally and by performing bus transactions on that bus only. For example, a write request by a processor needs only a single transaction to invalidate all copies of a block known to be shared locally. When a block is shared, but not known to be shared locally, a global invalidation must be used to invalidate all shared copies.

To benefit from local sharing, the protocol requires the local-shared-unmodified cache and directory states. Read requests are satisfied on the local bus when possible and indicate that the cached copy can go to the local shared state if this is so.

Root node. The exclusive-unmodified cache state for blocks on the root node allows reads and writes to proceed without bus transactions. This is critical for private data. The exclusive-unmodified state is not used for blocks cached on other nodes because of the need to inform the directory when the block is written and the cost associated with this.

In practice, a read miss by the root node to a block will be filled, and the cache state set to exclusive-unmodified. If the process then writes to the block, the cache state will change to exclusive-modified without taking a cache miss. This allows private data accesses to proceed without bus transactions.

Bus addresses in the directory. The protocol maintains the address of the bus on which a block resides when the block is owned exclusively or when it is shared locally. By keeping this location, the protocol greatly reduces the number of global invalidations by directing invalidations to this bus instead of falling back on global transactions. This reduces global invalidations to only those that are actually necessary and requires only a reasonable amount of space overhead in the directory. The amount of state required is just $\log_2(dn/p)$ bits per block.

The protocol forces transactions to notify the directory when a block changes from being cached on a single bus to being cached on multiple buses. ■

Acknowledgments

We would like to thank Jim Goodman of the University of Wisconsin-Madison for his helpful discussions. The research was partially sponsored by the Defense Advanced Research Projects Agency and monitored by the Office of Naval Research under Contract No. N00014-88-K-0579.

References

1. C.G. Bell, "Multis: A New Class of Multiprocessor Computer," *Science*, Vol. 228, Apr. 16, 1985, pp. 462-467.

2. J.R. Goodman and P.J. Woest, "The Wisconsin Multicube: A New Large-Scale Cache-Coherent Multiprocessor," *Proc. 15th Int'l Symp. Computer Architecture*, Computer Society Press, Los Alamitos, Calif., Order No. 861 (microfiche only), 1988, pp. 422-431.

3. P. Bitar and A. Despain, "Multiprocessor Cache Synchronization Issues, Innovations, Evolution," *Proc. 13th Int'l Symp. Computer Architecture*, Computer Society Press, Los Alamitos, Calif., Order No. 719 (microfiche only), 1986, pp. 424-433.

4. W.D. Weber and A. Gupta, "Analysis of Cache Invalidation Patterns in Microprocessors," *Proc. ASPLOS III*, Computer Society Press, Los Alamitos, Calif., Order No. 1936, 1989, pp. 243-256.

Michael Carlton is a computer science PhD candidate at the University of California at Berkeley. His research interests include cache coherency, highly parallel systems, and logic programming architectures.

Carlton received BS degrees with honors in information and computer sciences and mathematics from the University of California at Irvine in 1985. He is a student member of the IEEE Computer Society and ACM.

Alvin M. Despain was a professor of computer science at the University of California, Berkeley, until 1990. He is now a professor of electrical engineering systems and computer science at the University of Southern California. His research interests include computer architecture, multiprocessors and multicomputer systems, logic programming architectures, and design automation.

Despain received a BS degree in 1960, an MS in 1964, and a PhD in 1966, all in electrical engineering at the University of Utah. He is a member of IEEE, ACM, and AAAI.