The Directory-Based Cache Coherence Protocol for the DASH Multiprocessor

Daniel Lenoski, James Laudon, Kourosh Gharachorloo,
Anoop Gupta, and John Hennessy

Computer Systems Laboratory
Stanford University, CA 94305

Abstract

DASH is a scalable shared-memory multiprocessor currently being developed at Stanford's Computer Systems Laboratory. The architecture consists of powerful processing nodes, each with a portion of the shared-memory, connected to a scalable interconnection network. A key feature of DASH is its distributed directory-based cache coherence protocol. Unlike traditional snoopy coherence protocols, the DASH protocol does not rely on broadcast; instead it uses point-to-point messages sent between the processors and memories to keep caches consistent. Furthermore, the DASH system does not contain any single serialization or control point. While these features provide the basis for scalability, they also force a reevaluation of many fundamental issues involved in the design of a protocol. These include the issues of correctness, performance and protocol complexity. In this paper, we present the design of the DASH coherence protocol and discuss how it addresses the above issues. We also discuss our strategy for verifying the correctness of the protocol and briefly compare our protocol to the IEEE Scalable Coherent Interface protocol.

1 Introduction

The limitations of current uniprocessor speeds and the ability to replicate low cost, high-performance processors and VLSI components have provided the impetus for the design of multiprocessors which are capable of scaling to a large number of processors. Two major paradigms for these multiprocessor architectures have developed, *message-passing* and *shared-memory*. In a message-passing multiprocessor, each processor has a local memory, which is only accessible to that processor. Interprocessor communication occurs only through explicit message passing. In a shared-memory multiprocessor, all memory is accessible to each processor. The shared-memory paradigm has the advantage that the programmer is not burdened with the issues of data partitioning, and accessibility of data from all processors simplifies the task of dynamic load distribution. The primary advantage of the message passing systems is the ease with which they scale to support a large number of processors. For shared-memory machines providing such scalability has traditionally proved difficult to achieve.

We are currently building a prototype of a scalable shared-memory multiprocessor. The system provides high processor performance and scalability though the use of coherent caches and a directory-based coherence protocol. The high-level organization of the prototype, called DASH (Directory Architecture for SHared memory) [17], is shown in Figure 1. The architecture consists of a number of processing nodes connected through a high-bandwidth low-latency interconnection network. The physical memory in the machine is distributed among the nodes of the multiprocessor, with all memory accessible to each node. Each processing node, or *cluster*, consists of a small number of high-performance processors with their individual caches, a portion of the shared-memory, a common cache for pending remote accesses, and a directory controller interfacing the cluster to the network. A bus-based snoopy scheme is used to keep caches coherent within a cluster, while inter-node cache consistency is maintained using a distributed directory-based coherence protocol.

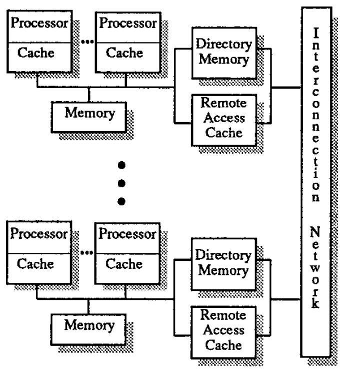


Figure 1: General architecture of DASH.

The concept of directory-based cache coherence was first proposed by Tang [20] and Censier and Feautrier [6]. Subsequently, it has been been investigated by others ([1],[2] and [23]). Building on this earlier work, we have developed a new directory-based cache-coherence protocol which works with distributed directories and the hierarchical cluster configuration of DASH. The protocol also integrates support for efficient synchronization operations using the directory. Furthermore, in designing the machine we have addressed many of the issues left unresolved by earlier work.

In DASH, each processing node has a directory memory corresponding to its portion of the shared physical memory. For each memory block, the directory memory stores the identities of all remote nodes caching that block. Using the directory memory, a node writing a location can send point-to-point invalidation or update messages to those processors that are actually caching that block. This is in contrast to the invalidating broadcast required by the snoopy protocol. The scalability of DASH depends on this ability to avoid broadcasts. Another important attribute of the directory-based protocol is that it does not depend on any specific interconnection network topology. As a result, we can readily use any of the low-latency scalable networks, such as meshes or hypercubes, that were originally developed for message-passing machines [7].

While the design of bus-based snoopy coherence protocols is reasonably well understood, this is not true of distributed directory-based protocols. Unlike snoopy protocols, directory-based schemes do not have a single serialization point for all memory transactions. While this feature is responsible for their scalability, it also makes them more complex and forces one to rethink how the protocol should address the fundamental issues of correctness, system performance, and complexity.

The next section outlines the important issues in designing a cache coherence protocol. Section 3 gives an overview of the DASH hardware architecture. Section 4 describes the design of the DASH coherence protocol, relating it to the issues raised in section 2. Section 5 outlines some of the additional operations supported beyond the base protocol, while Section 6 discusses scaling the directory structure. Section 7 briefly describes our approach to verifying the correctness of the protocol. Section 8 compares the DASH protocol with the proposed IEEE-SCI (Scalable Coherent Interface) protocol for distributed directory-based cache coherence. Finally, section 9 presents conclusions and summarizes the current status of the design effort.

2 Design Issues for Distributed Coherence Protocols

The issues that arise in the design of any cache coherence protocol and, in particular, a distributed directory-based protocol, can be divided into three categories: those that deal with correctness, those that deal with the performance, and those related to the distributed control of the protocol.

2.1 Correctness

The foremost issue that any multiprocessor cache coherence protocol must address is correctness. This translates into requirements in three areas:

Memory Consistency Model: For a uniprocessor, the model of a correct memory system is well defined. Load operations return the last value written to a given memory location. Likewise, store operations bind the value returned by subsequent loads of the location until the next store. For multiprocessors, however, the issue is more complex because the definitions of "last value written", "subsequent loads" and "next store" become less clear as there may be multiple processors reading and writing a location. To resolve this difficulty a number of memory consistency models have been proposed in the literature, most notably, the sequential and weak consistency models [8]. Weaker consistency models attempt to loosen the constraints on the coherence protocol while still providing a reasonable programming model for the user. Although most existing systems utilize a relatively strong consistency model, the larger latencies found in a distributed system favor the less constrained models.

Deadlock: A protocol must also be deadlock free. Given the arbitrary communication patterns and finite buffering within the memory system there are numerous opportunities for deadlock. For example, a deadlock can occur if a set of transactions holds network and buffer resources in a circular manner, and the consumption of one request requires the generation of another request. Similarly, lack of flow control in nodes can cause requests to back up into the network, blocking the flow of other messages that may be able to release the congestion.

Error Handling: Another issue related to correctness is support for data integrity and fault tolerance. Any large system will exhibit failures, and it is generally unacceptable if these failures result in corrupted data or incorrect results without a failure indication. This is especially true for parallel applications where algorithms are more complex and may contain some non-determinism which limits repeatability. Unfortunately, support for data integrity and fault-tolerance within a complex protocol that attempts to minimize latency and is executed directly by hardware is difficult. The protocol must attempt to balance the level of data integrity with the increase in latency and hardware complexity. At a minimum, the protocol should be able to flag all detectable failures, and convey this information to the processors affected.

2.2 Performance

Given a protocol that is correct, performance becomes the next important design criterion. The two key metrics of memory system performance are latency and bandwidth.

Latency: Performance is primarily determined by the latency experienced by memory requests. In DASH, support for cachable shared data provides the major reduction in latency. The latency of write misses is reduced by using write buffers and by the support of the release consistency model. Hiding the latency for read misses is usually more critical since the processor is stalled until data is returned. To reduce the latency for read misses, the protocol must minimize the number of inter-cluster messages needed to service a miss and the delay associated with each such message.

Bandwidth: Providing high memory bandwidth that scales with the number of processors is key to any large system. Caches and distributed memory form the basis for a scalable, high-bandwidth memory system in DASH. Even with distributed memory, however, bandwidth is limited by the serialization of requests in the memory system and the amount of traffic generated by each memory request.

Servicing a memory request in a distributed system often requires several messages to be transmitted. For example, a message to access a remote location generates a reply message containing the data, and possibly other messages invalidating remote caches. The component with the largest serialization in this chain limits the maximum throughput of requests. Serialization affects performance by increasing the queuing delays, and thus the latency, of memory requests. Queueing delays can become critical for locations that exhibit a large degree of sharing. A protocol should attempt to minimize the service time at all queuing centers. In particular, in a distributed system no central resources within a node should be blocked while inter-node communication is taking place to service a request. In this way serialization is limited only by the time of local, intra-node operations.

The amount of traffic generated per request also limits the effective throughput of the memory system. Traffic seen by the global interconnect and memory subsystem increases the queueing for these shared resources. DASH reduces traffic by providing coherent caches and by distributing memory among the processors. Caches filter many of the requests for shared data while grouping memory with processors removes private references if the corresponding memory is allocated within the local cluster. At the protocol level, the number of messages required to service different types of memory requests should be minimized, unless the extra messages directly contribute to reduced latency or serialization.

2.3 Distributed Control and Complexity

A coherence protocol designed to address the above issues must be partitioned among the distributed components of the multiprocessor. These components include the processors and their caches, the directory and main memory controllers, and the interconnection network. The lack of a single serialization point, such as a bus, complicates the control since transactions do not complete atomically. Furthermore, multiple paths within the memory system and lack of a single arbitration point within the system allow some operations to complete out of order. The result is that there is a rich set of interactions that can take place between different memory and coherence transactions. Partitioning the control of the protocol requires a delicate balance between the performance of the system and the complexity of the components. Too much complexity may effect the ability to implement the protocol or ensure that the protocol is correct.

3 Overview of DASH

Figure 2 shows a high-level picture of the DASH prototype we are building at Stanford. In order to manage the size of the prototype design effort, a commercial bus-based multiprocessor was chosen as the processing node. Each node (or *cluster*) is a Silicon Graphics POWER Station 4D/240 [4]. The 4D/240 system consists of four high-performance processors, each connected to a 64 Kbyte first-level instruction cache, and a 64 Kbyte write-through data cache. The 64 Kbyte data cache interfaces to a 256 Kbyte second-level write-back cache through a read buffer and a 4 word deep write-buffer. The main purpose of this second-level cache is to convert the write-through policy of the first-level to a write-back policy, and to provide the extra cache tags for bus snooping. Both the first and second-level caches are direct-mapped.

In the 4D/240, the second-level caches are responsible for bus snooping and maintaining consistency among the caches in the cluster. Consistency is maintained using the Illinois coherence protocol [19], which is an invalidation-based ownership protocol. Before a processor can write to a cache line, it must first acquire exclusive ownership of that line by requesting that all other caches invalidate their copy of that line. Once a processor has exclusive ownership of a cache line, it may write to that line without consuming further bus cycles.

The memory bus (MPBUS) of the 4D/240 is a pipelined synchronous bus, supporting memory-to-cache and cache-to-cache transfers of 16 bytes every 4 bus clocks with a latency of 6 bus clocks. While the MPBUS is pipelined, it is not a split transaction bus. Consequently, it is not possible to efficiently interleave long duration remote transactions with the short duration local

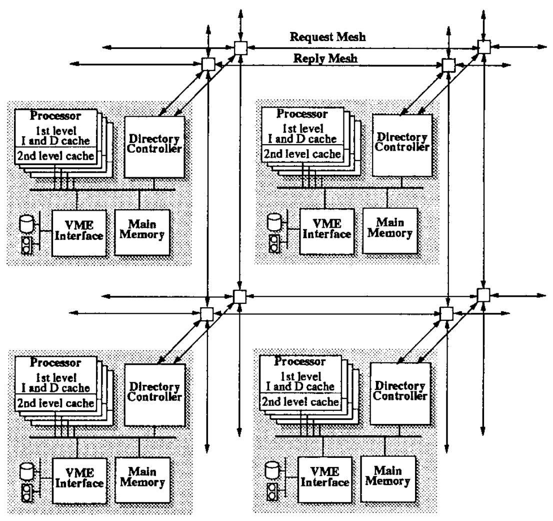


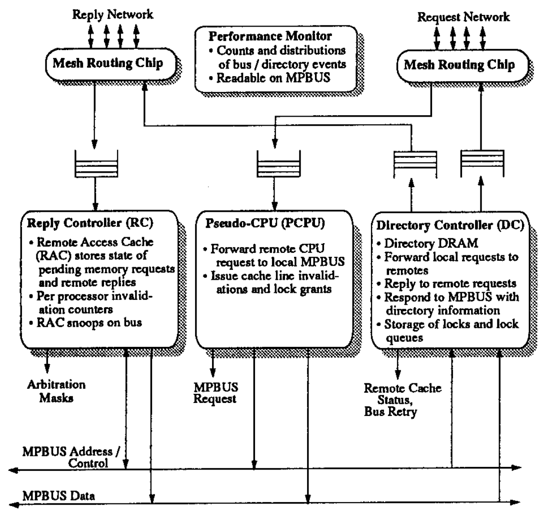
Figure 2: Block diagram of sample 2 x 2 DASH system.

transactions. Since this ability is critical to DASH, we have extended the MPBUS protocol to support a retry mechanism. Remote requests are signaled to retry while the inter-cluster messages are being processed. To avoid unnecessary retries the processor is masked from arbitration until the response from the remote request has been received. When the response arrives, the requesting processor is unmasked, retries the request on the bus, and is supplied the remote data.

A DASH system consists of a number of modified 4D/240 systems that have been supplemented with a directory controller board. This directory controller board is responsible for maintaining the cache coherence across the nodes and serving as the interface to the interconnection network.

The directory board is implemented on a single printed circuit board and consists of five major subsystems as shown in Figure 3. The *directory controller* (DC) contains the directory memory corresponding to the portion of main memory present within the cluster. It also initiates out-bound network requests and replies. The *pseudo-CPU* (PCPU) is responsible for buffering incoming requests and issuing such requests on the cluster bus. It mimics a CPU on this bus on behalf of remote processors except that responses from the bus are sent out by the directory controller. The *reply controller* (RC) tracks outstanding requests made by the local processors and receives and buffers the corresponding replies from remote clusters. It acts as memory when the local processors are allowed to retry their remote requests. The *network interface* and the local portion of the network itself reside on the directory card. The interconnection network consists of a pair of meshes. One mesh is dedicated to the request messages while the other handles replies. These meshes utilize *wormhole routing* [9] to minimize latency. Finally, the board contains *hardware monitoring logic* and miscellaneous control and status registers. The monitoring logic samples a variety of directory board and bus events from which usage and performance statistics can be derived.

The directory memory is organized as an array of directory



Reply Network
Mesh Routing Chip
Performance Monitor
• Counts and distributions of bus / directory events
• Readable on MPBUS
Request Network
Mesh Routing Chip
Reply Controller (RC)
• Remote Access Cache (RAC) stores state of pending memory requests and remote replies
• Per processor invalidation counters
• RAC snoops on bus
Pseudo-CPU (PCPU)
• Forward remote CPU request to local MPBUS
• Issue cache line invalidations and lock grants
Directory Controller (DC)
• Directory DRAM
• Forward local requests to remotes
• Reply to remote requests
• Respond to MPBUS with directory information
• Storage of locks and lock queues
Arbitration Masks
MPBUS Request
Remote Cache Status, Bus Retry
MPBUS Address / Control
MPBUS Data

Figure 3: Directory board block diagram.

entries. There is one entry for each memory block. The directory entries used in the prototype are identical to that originally proposed in [6]. They are composed of a single state bit together with a bit vector of pointers to clusters. The state bit indicates whether the clusters have a read (shared) or read/write (dirty) copy of the data. The bit vector contains a bit for each of the sixteen clusters supported in the prototype. Associating the directory with main memory allows the directory to be built with the same DRAM technology as main memory. The DC accesses the directory memory on each MPBUS transaction along with the access to main memory. The directory information is combined with the type of bus operation, address, and result of the snooping within the cluster to determine what network messages and bus controls the DC will generate.

The RC maintains its state in the *remote access cache* (RAC). The functions of the RAC include maintaining the state of currently outstanding requests, buffering replies from the network and supplementing the functionality of the processors' caches. The RAC is organized as a snoopy cache with augmented state information. The RAC's state machines allow accesses from both the network and the cluster bus. Replies from the network are buffered in the RAC and cause the waiting processor to be released for bus arbitration. When the released processor retries the access the RAC supplies the data via a cache-to-cache transfer.

3.1 Memory Consistency in DASH

As stated in Section 2, the correctness of the coherence protocol is a function of the memory consistency model adopted by the architecture. There is a whole spectrum of choices for the level of consistency to support directly in hardware. At one end is the *sequential consistency* model [16] which requires the execution of the parallel program to appear as some interleaving of the execution of the parallel processes on a sequential machine. As one moves towards weaker models of consistency, performance gains are made at the cost of a more complex programming model for the user.

The base model of consistency provided by the DASH hardware is called *release consistency*. Release consistency [10] is an extension of the weak consistency model first proposed by Dubois, Scheurich and Briggs [8]. The distinguishing characteristics of release consistency is that it allows memory operations issued by a given processor to be observed and complete out of order with respect to the other processors. The ordering of operations is only preserved before "releasing" synchronization operations or explicit ordering operations. Release consistency takes advantage of the fact that while in a critical region a programmer has already assured that no other processor is accessing the protected variables. Thus, updates to these variables can be observed by other processors in arbitrary order. Only before the lock release at the end of the region does the hardware need to guarantee that all operations have completed. While release consistency does complicate programming and the coherence protocol, it can hide much of the overhead of write operations.

Support for release consistency puts several requirements on the system. First, the hardware must support a primitive which guarantees the ordering of memory operations at specific points in a program. Such fence [5, 10] primitives can then be placed by software before releasing synchronization points in order to implement release consistency. DASH supports two explicit fence mechanisms. A *full-fence* operation stalls the processor until all of its pending operations have been completed, while a *write-fence* simply delays subsequent write-operations. A higher performance implementation of release consistency includes implicit fence operations within the releasing synchronization operations themselves. DASH supports such synchronization operations yielding release consistency as its base consistency model. The explicit fence operations in DASH then allow the user or compiler to synthesize stricter consistency models if needed.

The release consistency model also places constraints on the base coherence protocol. First, the system must respect the local dependencies generated by the memory operations of a single processor. Second, all coherence operations, especially operations related to writes, must be acknowledged so that the issuing processor can determine when a fence can proceed. Third, any cache line owned with pending invalidations against it can not be shared between processors. This prevents the new processor from improperly passing a fence. If sharing is allowed then the receiving processor must be informed when all of the pending invalidates have been acknowledged. Lastly, any operations that a processor issues after a fence operation may not become visible to any other processor until all operations preceding the fence have completed.

4 The DASH Cache Coherence Protocol

In our discussion of the coherence protocol, we use the following naming conventions for the various clusters and memories involved in any given transaction. A *local cluster* is a cluster that contains the processor originating a given request, while the *home cluster* is the cluster which contains the main memory and directory for a given physical memory address. A *remote cluster* is any other cluster. Likewise, *local memory* refers to the main memory associated with the local cluster while *remote memory* is any memory whose home is not the local.

The DASH coherence protocol is an invalidation-based ownership protocol. A memory block can be in one of three states as indicated by the associated directory entry: (i) *uncached-remote*, that is not cached by any remote cluster; (ii) *shared-remote*, that is cached in an unmodified state by one or more remote clusters; or (iii) *dirty-remote*, that is cached in a modified state by a single remote cluster. The directory does not maintain information concerning whether the home cluster itself is caching a memory block because all transactions that change the state of a memory block are issued on the bus of the home cluster, and the snoopy bus protocol keeps the home cluster coherent. While we could have chosen not to issue all transactions on the home cluster's bus this would had an insignificant performance improvement since most requests to the home also require an access to main memory to retrieve the actual data.

The protocol maintains the notion of an *owning cluster* for each memory block. The owning cluster is nominally the home cluster. However, in the case that a memory block is present in the dirty state in a remote cluster, that cluster is the owner. Only the owning cluster can complete a remote reference for a given block and update the directory state. While the directory entry is always maintained in the home cluster, a dirty cluster initiates all changes to the directory state of a block when it is the owner (such update messages also indicate that the dirty cluster is giving up ownership). The order that operations reach the owning cluster determines their global order.

As with memory blocks, a cache block in a processor's cache may also be in one of three states: invalid, shared, and dirty. The shared state implies that there may be other processors caching that location. The dirty state implies that this cache contains an exclusive copy of the memory block, and the block has been modified.

The following sections outline the three primitive operations supported by the base DASH coherence protocol: read, read-exclusive and write-back. We also discuss how the protocol responds to the issues that were brought up in Section 2 and some of the alternative design choices that were considered. We describe only the normal flow for the memory transactions in the following sections, exception cases are covered in section 4.6.

4.1 Read Requests

Memory read requests are initiated by processor load instructions. If the location is present in the processor's first-level cache, the cache simply supplies the data. If not present, then a cache fill operation must bring the required block into the first-level cache. A fill operation first attempts to find the cache line in the processor's second-level cache, and if unsuccessful, the processor issues a read request on the bus. This read request either completes locally or is signaled to retry while the directory board interacts with the other clusters to retrieve the required cache line. The detailed flow for a read request is given in Figure 7 in the appendix.

The protocol tries to minimize latency by using cache-to-cache transfers. The local bus can satisfy a remote read if the given line is held in another processor's cache or the remote access cache (RAC). The four processor caches together with the RAC form a five-way set associative (1.25 Mbyte) cluster cache. The effective size of this cache is smaller than a true set associative cache because the entries in the caches need not be distinct. The check for a local copy is initiated by the normal snooping when the read is issued on the bus. If the cache line is present in the shared state then the data is simply transferred over the bus to the requesting processor and no access to the remote home cluster is needed. If the cache line is held in a dirty state by a local processor, however, something must be done with the ownership of the cache line since the processor supplying the data goes to a shared state in the Illinois protocol used on the cluster bus. The two options considered were to: (i) have the directory do a sharing write-back to the home cluster; and (ii) have the RAC take ownership of the cache line. We chose the second option because it permits the processors within a cluster to read and write a shared location without causing traffic in the network or home cluster.

If a read request cannot be satisfied by the local cluster, the processor is forced to retry the bus operation, and a request message is sent to the home cluster. At the same time the processor is masked from arbitration so that it does not tie up the local bus. Whenever a remote request is sent by a cluster, a RAC entry is allocated to act as a placeholder for the reply to this request. The RAC entry also permits merging of requests made by the different processors within the same cluster. If another request to the same memory block is made, a new request will not be sent to the home cluster; this reduces both traffic and latency. On the other hand, an access to a different memory block, which happens to map to a RAC entry already in use, must be delayed until the pending operation is complete. Given that the number of active RAC entries is small the benefit of merging should outweigh the potential for contention.

When the read request reaches the home cluster, it is issued on that cluster's bus. This causes the directory to look up the status of that memory block. If the block is in an uncached-remote or shared-remote state the directory controller sends the data over the reply network to the requesting cluster. It also records the fact that the requesting cluster now has a copy of the memory block. If the block is in the dirty-remote state, however, the read request is forwarded to the owning, dirty cluster. The owning cluster sends out two messages in response to the read. A message containing the data is sent directly to the requesting cluster, and a sharing writeback request is sent to the home cluster. The sharing writeback request writes the cache block back to memory and also updates the directory. The flow of messages for this case is shown in Figure 4.

As shown in Figure 4, any request not satisfied in the home cluster is forwarded to the remote cluster that has a dirty copy of the data. This reduces latency by permitting the dirty cluster to respond directly to the requesting cluster. In addition, this forwarding strategy allows the directory controller to simultaneously process many requests (i.e. to be multithreaded) without the added complexity of maintaining the state of outstanding requests. Serialization is reduced to the time of a single intra-cluster bus transaction. The only resource held while inter-cluster messages are being sent is a single entry in the originating cluster's RAC.

The downside of the forwarding strategy is that it can result in additional latency when simultaneous accesses are made to the same block. For example, if two read requests from different clusters are received close together for a line that is dirty remote, both will be forwarded to the dirty cluster. However, only the first one will be satisfied since this request will force the dirty cluster to lose ownership by doing a sharing writeback and changing its local state to read only. The second request will not find the dirty data and will be returned with a *negative acknowledge* (NAK) to its originating cluster. This NAK will force the cluster to retry its access. An alternative to the forwarding approach used by our protocol would have been to buffer the read request at the home cluster, have the home send

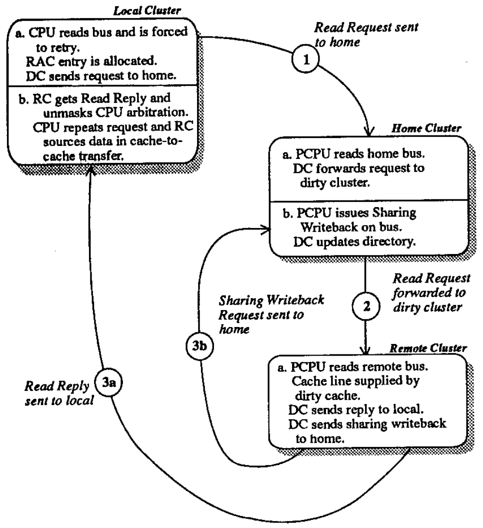

Local Cluster
a. CPU reads bus and is forced to retry. RAC entry is allocated. DC sends request to home.
b. RC gets Read Reply and unmasks CPU arbitration. CPU repeats request and RC sources data in cache-to-cache transfer.
Read Request sent to home
1
Home Cluster
a. PCPU reads home bus. DC forwards request to dirty cluster.
b. PCPU issues Sharing Writeback on bus. DC updates directory.
Read Request forwarded to dirty cluster
2
Sharing Writeback Request sent to home
3b
Remote Cluster
a. PCPU reads remote bus. Cache line supplied by dirty cache. DC sends reply to local. DC sends sharing writeback to home.
Read Reply sent to local
3a

Figure 4: Flow of Read Request to remote memory with directory in dirty-remote state.

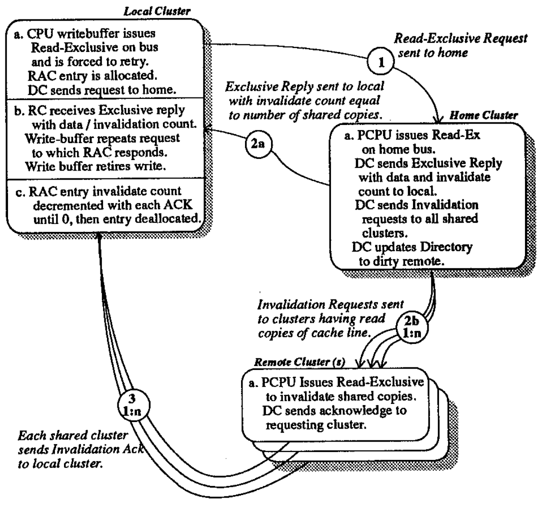

Local Cluster
a. CPU writebuffer issues Read-Exclusive on bus and is forced to retry. RAC entry is allocated. DC sends request to home.
b. RC receives Exclusive reply with data / invalidation count. Write-buffer repeats request to which RAC responds. Write buffer retires write.
c. RAC entry invalidate count decremented with each ACK until 0, then entry deallocated.
Read-Exclusive Request sent to home
1
Exclusive Reply sent to local with invalidate count equal to number of shared copies.
2a
Home Cluster
a. PCPU issues Read-Ex on home bus. DC sends Exclusive Reply with data and invalidate count to local. DC sends Invalidation requests to all shared clusters. DC updates Directory to dirty remote.
Invalidation Requests sent to clusters having read copies of cache line.
2b 1:n
Remote Cluster (s)
a. PCPU Issues Read-Exclusive to invalidate shared copies. DC sends acknowledge to requesting cluster.
3 1:n
Each shared cluster sends Invalidation Ack to local cluster.

Figure 5: Flow of Read-Exclusive Request to remote memory with directory in shared-remote state.

a flush request to the owning cluster, and then have the home send the data back to the originating cluster. We did not adopt this approach because it would have increased the latency for such reads by adding an extra network and bus transaction. Additionally, it would have required buffers in the directory to hold the pending transaction, or blocking subsequent accesses to the directory until the first request had been satisfied.

4.2 Read-Exclusive Requests

Write operations are initiated by processor store instructions. Data is written through the first-level cache and is buffered in a four word deep write-buffer. The second-level cache can retire the write if it has ownership of the line. Otherwise, a read-exclusive request is issued to the bus to acquire sole ownership of the line and retrieve the other words in the cache block. Obtaining ownership does not block the processor directly; only the write-buffer output is stalled. As in the case of read requests, cache coherence operations begin when the read-exclusive request is issued on the bus. The detailed flow of read-exclusive request is given in the appendix in Figure 9 and is summarized below.

The flow of a read-exclusive is similar to that of a read request. Once the request is issued on the bus, it checks other caches at the local cluster level. If one of those caches has that memory block in the dirty state (it is the owner), then that cache supplies the data and ownership and invalidates its own copy. If the memory block is not owned by the local cluster, a request for ownership is sent to the home cluster. As in the case of read requests, a RAC entry is allocated to receive the ownership and data.

At the home cluster, the read-exclusive request is echoed on the bus. If the memory block is in an uncached-remote or shared-remote state the data and ownership are immediately sent back over the reply network. In addition, if the block is in the shared-remote state, each cluster caching the block is sent an invalidation request. The requesting cluster receives the data as before, and is also informed of the number of invalidation acknowledge messages to expect. Remote clusters send invalidation acknowledge messages to the requesting cluster after completing their invalidation. As discussed in Section 3.1, the invalidation acknowledges are needed by the requesting processor to know when the store has been completed with respect to all processors. The RAC entry in the requesting cluster persists until all invalidation acknowledges have been received. The receipt of the acknowledges generally occurs after the processor itself has been granted exclusive ownership of the cache line and continued execution. Figure 5 depicts this shared-remote case.

If the directory indicates a dirty-remote state, then the request is forwarded to the owning cluster as in a read request. At the dirty cluster, the read-exclusive request is issued on the bus. This causes the owning processor to invalidate that block from its cache and to send a message to the requesting cluster granting ownership and supplying the data. In parallel, a request is sent to the home cluster to update ownership of the block. On receiving this message, the home sends an acknowledgment to the new owning cluster. This extra acknowledgment is needed because the requesting cluster (the new owning cluster) may give up ownership (e.g. due to a writeback) even before the home directory has received an ownership change message from the previous owner. If these messages reach the home out of order the directory will become permanently inconsistent. The extra acknowledgment guarantees that the new owner retain ownership until the directory has been updated.

Performance of the read and write operations is closely related to the speed of the MPBUS and the latency of inter-cluster communication. Figure 6 shows the latencies for various mem-

Read Operations	
Hit in 1st Level Cache	1 pclock
Fill from 2nd Level Cache	12 pclock
Fill from Local Cluster	22 pclock
Fill from Remote Cluster	61 pclock
Fill from Dirty Remote, Remote Home	80 pclock
Fill operations fetch 16 byte cache blocks and empty the write-buffer before fetching the read-miss cache block.	
Write Operations	
Hit on 2nd Level Owned Block	3 pclock
Owned by Local Cluster	18 pclock
Owned in Remote Cluster	57 pclock
Owned in Dirty Remote, Remote Home	76 pclock
Write operations only stall the write-buffer, not the processor, while the fill is outstanding. Write delays assume Release Consistency (i.e. they do not wait for remote invalidates to be acknowledged).	

Figure 6: Latency for various memory system operations in processor clocks. Each processor clock in the prototype is 40 ns.

ory operations in the DASH prototype assuming no network or bus contention. The figure illustrates the one-to-one relationship between the latency of an operation and its corresponding number of network hops and bus transactions. In DASH, the network and directory board overhead is roughly equal to the CPU overhead to initiate a bus transaction. Thus, if an intra-cluster bus transaction takes roughly 20 processor clocks then an inter-cluster transaction that involves two clusters, (i.e. three bus transactions) takes roughly 60 processor clocks, and a three cluster transaction takes 80 processor clocks.

4.3 Writeback Requests

A dirty cache line that is replaced must be written back to memory. If the home of the memory block is the local cluster, then the data is simply written back to main memory. If the home cluster is remote, then a message is sent to the remote home which updates the main memory and marks the block uncached-remote. The flow of a writeback operation is given in the appendix in Figure 8.

4.4 Bus Initiated Cache Transactions

CPU initiated transactions have been described in the preceding sections. The protocol also includes transitions made by the slave caches that are monitoring their respective buses. These transitions are equivalent to those in a normal snoopy bus protocol. In particular, a read operation on the bus will cause a dirty cache to supply data and change to a shared state. Dirty data will also be written back to main memory (or the RAC if remote). A read-exclusive operation on the bus will cause all other cached copies of the line to be invalidated. Note that when a valid line in the second-level cache is invalidated, the first-level cache is also invalidated so that the processor's second-level cache is a superset of the first-level cache.

4.5 Support for Memory Consistency

As discussed in section 3.1, DASH supports the release consistency model. Memory system latency is reduced because the semantics of release consistency allows the processor to continue after issuing a write operation. The write-buffer within the processor holds the pending operation, and the write-buffer is allowed to retire the write before the operation has completed with respect to all processors. The processor itself is allowed to continue while the write-buffer and directory controller are completing the previous operations. Ordering of memory accesses is only guaranteed between operations separated by a releasing synchronization operation or an explicit fence operation. Upon a write-fence (explicit or implicit), all previous read and write operations issued by this processor must have completed with respect to all processors before any additional write operations can become visible to other processors.

DASH implements a write fence by blocking a processor's access to its second-level cache and the MPBUS until all reads and writes it issued before the write fence have completed. This is done by stalling the write-fence (which is mapped to a store operation) in the processor's write-buffer. Guaranteeing that preceding reads and writes have been performed without imposing undue processor stalls is the challenge. A first requirement is that all invalidation operations must be acknowledged. As illustrated in Figure 5, a write operation to shared data can proceed after receiving the exclusive reply from the directory, but the RAC entry associated with this operation persists until all of the acknowledges are received by the reply controller (RC). Each RAC entry is tagged with the processor that is responsible for this entry and each processor has a dedicated counter in the RC which counts the total number of RAC entries in use by that processor. A write fence stalls until the counter for that processor is decremented to zero. At this point, the processor has no outstanding RAC entries, so all of its invalidation acknowledges must have been received.

We observe that simply using a per processor counter to keep track of the number of outstanding invalidations is not sufficient to support release consistency. A simple counter does not allow the processor cache to distinguish between dirty cache lines that have outstanding invalidates from those that do not. This results in another processor not being able to detect whether a line returned by a dirty cache has outstanding invalidates. The requesting processor could then improperly pass through a fence operation. Storing the pending invalidate count on a per cache line basis in the RAC, and having the RAC snoop bus transactions, allows cache lines with pending invalidates to be distinguished. The RAC forces a reject of remote requests to such blocks with a NAK reply. Local accesses are allowed, but the RAC adds the new processor to its entry for the line making this processor also responsible for the original invalidations. Write-back requests of a line with outstanding invalidations are blocked by having the RAC take dirty ownership of the cache block.

In the protocol, invalidation acknowledges are sent to the local cluster that initiated the memory request. An alternative would be for the home cluster to gather the acknowledges, and, when all have been received, send a message to the requesting cluster indicating that the request has been completed. We chose the former because it reduces the waiting time for completion of a subsequent fence operation by the requesting cluster and reduces the potential of a hot spot developing at the memory.

4.6 Exception Conditions

The description of the protocol listed above does not cover all of the conditions that the actual protocol must address. While enumerating all of the possible exceptions and protocol responses would require an overly detailed discussion, this section introduces most of the exception cases and gives an idea of how the protocol responds to each exception.

One exception case is that a request forwarded to a dirty cluster may arrive there to find that the dirty cluster no longer owns the data. This may occur if another access had previously been forwarded to the dirty cluster and changed the ownership of the block, or if the owning cluster performs a writeback. In these cases, the originating cluster is sent a NAK response and is required to reissue the request. By this time ownership should have stabilized and the request will be satisfied. Note that the reissue is accomplished by simply releasing the processor's arbitration mask and treating this as a new request instead of replying with data.

In very pathological cases, for example when ownership for a block is bouncing back and forth between two remote clusters, a requesting cluster (some third cluster) may receive multiple NAK's and may eventually time-out and return a bus error. While this is undesirable, its occurrence is very improbable in the prototype system and, consequently, we do not provide a solution. In larger systems this problem is likely to need a complete answer. One solution would be to implement an additional directory state which signifies that other clusters are queued for access. Only the first access for a dirty line would be forwarded while this request and subsequent requests are queued in the directory entry. Upon receipt of the next ownership change the directory can respond to all of the requests if they are for read only copies. If some are for exclusive access then ownership can be granted to each in turn on a pseudo-random basis. Thus, eventually all requests will be fulfilled.

Another set of exceptions arise from the multiple paths present in the system. In particular, the separate request and reply networks together with their associated input and output FIFO's and bus requesters imply that some messages sent between two clusters can be received out of order. The protocol can handle most of these misorderings because operations are acknowledged and out-of-order requests simple receive NAK responses. Other cases require more attention. For example, a read reply can be overtaken by an invalidate request attempting to purge the read copy. This case is handled by the snooping on the RAC. When the RAC sees an invalidation request for a pending read, it changes the state of that RAC entry to invalidated-read-pending. In this state, the RC conservatively assumes that any read reply is stale and treats the reply as a NAK response.

4.7 Deadlock

In the DASH prototype, deadlocks are eliminated through a combination of hardware and protocol features. At the hardware level, DASH consists of two mesh networks, each of which guarantees point-to-point delivery of messages without deadlocks. However, this by itself is not sufficient to prevent deadlocks because the consumption of an incoming message may require the generation of another outgoing message. This can result in circular dependencies between the limited buffers present in two or more nodes and cause deadlock.

To address this problem, the protocol divides all messages into request messages (e.g. read and read-exclusive requests and invalidation requests) and reply messages (e.g. read and read-exclusive replies and invalidation acknowledges). Furthermore, one mesh is dedicated to servicing request messages while the other handles reply messages. Reply messages are guaranteed to be consumed at the destination, partly because of their nature and partly because space for the reply data is preallocated in the RAC. This eliminates the possibility of request-reply circular dependencies and the associated deadlocks.

However, the protocol also relies on request messages that generate additional requests. Because of the limited buffer space, this can result in deadlocks due to request-request circular dependencies. Fairly large input and output FIFO's reduce the probability of this problem. If it does arise, the directory hardware includes a time-out mechanism to break the possible deadlock. If the directory has been blocked for more than the time-out period in attempting to forward a request it will instead reject the request with a NAK reply message. Once this deadlock breaking mode is entered enough other requests are handled similarly so that any possible deadlock condition that has arisen within the request network can be eliminated. As in cases discussed earlier, this scheme relies on the processor's ability to reissue its request upon receiving a NAK.

4.8 Error Handling

The final set of exceptions arise in response to error conditions in the hardware or protocol. The system includes a number of error checks including ECC on main memory, parity on the directory memory, length checking of network messages and inconsistent bus and network message checking. These checks are reported to processors through bus errors and associated error capture registers. Network errors and improper requests are dropped by the receiver of such messages. Depending upon the type of network message that was lost or corrupted, the issuing processor will eventually time-out its originating request or some fence operation which will be blocked waiting for a RAC entry to be deallocated. The time-out generates a bus-error which interrupts the processor. The processes using the particular memory location are aborted, but low level operating system code can recover from the error if it is not within the kernel. The OS can subsequently clean up the state of a line by using back-door paths that allow direct addressing of the RAC and directory memory.

5 Supplemental Operations

During the evolution of the DASH protocol, several additional memory operations were evaluated. Some of these operations are included in the DASH prototype, while others were not included due to hardware constraints or a lack of evidence that the extension would provide significant performance gains.

The first major extension incorporated into the DASH protocol was support for synchronization operations. The sharing characteristics of synchronization objects are often quite different from those of normal data. Locks, barriers, and semaphores can be highly contended. Using the normal directory protocol for synchronization objects can lead to hot spots. For example, when a highly contended lock is released, all processor caches containing the lock are invalidated; this invalidation results in the waiting processors rushing to grab the lock. DASH provides special *queue-based lock* primitives that use the directory memory to keep track of clusters waiting for a lock. Using the directory memory is natural since it is already set up to track queued clusters, and the directory is normally accessed in read-modify-write cycles that match the atomic update necessary for locks. An unlock of a queue-based lock while clusters are waiting results in a grant of the lock being sent to one of the waiting clusters. This grant allows the cluster to obtain the lock without any further network messages. Thus, queue-based locks reduce the hot spotting generated by contended locks and reduce the latency between an unlock operation and subsequent acquisition of the lock. This and other synchronization primitives are discussed in detail in [17].

Another set of operations included in the prototype help hide the latency of memory operations. Normally, when a read is issued the processor is stalled until the data comes back. With very fast processors, this latency can be tens to hundreds of processor cycles. Support for some form of prefetch can clearly help. DASH supports both *read prefetch* and *read-exclusive prefetch* operations [17]. These operations cause the directory controller to send out a read or read-exclusive request for the data, but do not block the processor. Thus, the processor is able to overlap the fetching of the data with useful work. When the processor is ready to use the prefetched data, it issues a normal read or read exclusive request. By this time the data will either be in the RAC or the prefetch will be outstanding, in which case the normal read or read-exclusive is merged with the prefetch. In either case, the latency for the data will be reduced. Ideally, we would have liked to place the prefetched data directly in the requesting processor's cache instead of the RAC, but that would have required significant modifications to the existing processor boards.

There are some variables for which a write-update coherence protocol is more appropriate than the DASH write-invalidate protocol [3]. The prototype system provides for a single word *update write* primitive which updates memory and all the caches currently holding the word. Since exclusive ownership is not required, the producer's write buffer can retire the write as soon as it has been issued on the bus. Update-writes are especially useful for event synchronization. The producer of an event can directly update the value cached by the waiting processor reducing the latency and traffic that would result if the value was invalidated. This primitive is especially useful in implementing barriers, as an update-write can be used by the last processor entering the barrier to release all waiting processors. Update operations conform to the release consistency memory model, but require explicit fence operations when used for synchronization purposes.

6 Scalability of the DASH Directory

The DASH directory scheme currently uses a full bit-vector to identify the remote clusters caching a memory block. While this is reasonable for the DASH prototype, it does not scale well since the amount of directory memory required is the proportional to the product of the main memory size and the number of processors in the system. We are currently investigating a variety of solutions which limit the overhead of directory memory. The most straightforward modification is the use of a limited number of pointers per directory entry. Each directory pointer holds the cluster number of a cluster currently caching the given line. In any limited pointer scheme some mechanism must exist to handle cache blocks that are cached by more processors then there are pointers. A very simple scheme resorts to a broadcast in these cases [1]. Better results can be obtained if the pointer storage memory reverts to a bit vector when pointer overflow occurs. Of course, a complete bit vector is not possible, but if each bit represents a *region* of processors the amount of traffic generated by such overflows can be greatly reduced relative to a broadcast.

Other schemes to scale the directory rely on restructuring of directory storage. Possible solutions include allowing pointers to be shared between directory entries, or using a cache of directory entries to supplement or replace the normal directory [18, 13]. A directory structured as a cache need not have a complete backing memory since replaced directory entries can simply invalidate their associated cache entries (similar to how multi-level caches maintain their inclusion property). Recent studies [13] have shown that such *sparse-directories* can maintain a constant overhead of directory memory compared with a full-bit vector when the number of processors grows from 64 to 1024. A sparse directory using limited pointers and a coarse vector only increases the total traffic by only 10-20% and should have minimal impact on processor performance. Furthermore, such directory structures require only small changes to the coherence protocol given here.

7 Validation of the Protocol

Validation of the DASH protocol presents a major challenge. Each cluster in DASH contains a complex directory controller with a large amount of state. This state coupled with the distributed nature of the DASH protocol results in an enormous number of possible interactions between the controllers. Writing a test suite that exercises all possible interactions in reasonable time seems intractable. Therefore, we are using two less exhaustive testing methods. Both these methods rely on the software simulator of DASH that we have developed.

The simulator consists of two tightly coupled components: a low-level DASH system simulator that incorporates the coherence protocol, and simulates the processor caches, buses, and interconnection network at a very fine level of detail; and Tango [11], a high-level functional simulator that models the processors and executes parallel programs. Tango simulates parallel processing on a uniprocessor while the DASH simulator provides detailed timing about latency of memory references. Because of the tight coupling between the two parts, our simulator closely models the DASH machine.

Our first scheme for testing the protocol consists of running existing parallel programs for which the results are known and comparing the output with that from the DASH simulator. The drawback of using parallel programs to check the protocol is that they use the memory system and synchronization features in "well-behaved" ways. For example, a well-written parallel program will not release a lock that is already free, and parallel programs usually don't modify shared variables outside of a critical section. As a result, parallel applications do not test a large set of possible interactions.

To get at the more pathological interactions, our second method relies on test scripts. These scripts can be written to provide a fine level of control over the protocol transitions and to be particularly demanding of the protocol. While writing an exhaustive set of such test scripts is not feasible, we hope to achieve reasonable test coverage with a smaller set of scripts by introducing randomness into the execution of the scripts.

The randomness idea used is an extension of the Berkeley Random Case Generation (RCG) technique [22] used to verify the SPUR cache controller design. Our method, called Intelligent Case Generation (ICG), is described in detail in [14]. Each script is a self-contained test sequence which executes a number of memory operations on a set of processors. Each script consists of some initialization, a set of test operations, and a check for proper results. Like RCG, multiple, independent scripts run simultaneously and interact in two ways. First, a processor randomly chooses which of the multiple active scripts it is going to pick its next action from. Therefore, execution of the same set of scripts will be interleaved in time differently upon each run. Second, while each script uses unique memory locations, these locations may be in the same cache line. Scripts interact by changing the cache state of cache lines used by other scripts.

ICG extends RCG in three ways. First, instead of simple two step scripts (a write followed by a read), ICG supports multi-step scripts in which some steps are executed in series and some are allowed to execute in parallel. Second, ICG provides a finer level of control over which processors execute which steps of a script and introduces randomness into the assignment process. Finally, ICG allows for a more flexible assignment of test addresses so that particular scripts do not have to be written to interact. Using ICG to dynamically assign addresses results in different scripts interacting at different times during a run, and results in the same script using various combinations of local and remote memory.

Of course, the hardware itself will also serve as a verification tool. The hardware can run both parallel programs and test scripts. While debugging protocol errors on the hardware will be difficult, the sheer number of cycles executed will be a demanding test of the protocol.

8 Comparison with Scalable Coherent Interface Protocol

Several protocols that provide for distributed directory-based cache coherence have been proposed [15, 21]. The majority of these protocols have not been defined in enough detail to do a reasonable comparison with the DASH protocol. One exception is the IEEE P1596 - Scalable Coherent Interface (SCI) [12]. While still evolving, SCI has been documented in sufficient detail to make a comparison possible. SCI differs from DASH, however, in that it is only an interface standard, not a complete system design. SCI only specifies the interfaces that each processor should implement, leaving open the actual node design and exact interconnection network.

At the system level, a typical SCI system would be similar to DASH with each processing node containing a processor, a section of main memory, and an interface to the interconnection network. Both systems rely on coherent caches maintained by distributed directories and distributed memories to provide scalable memory bandwidth. The major difference lies in how and where the directory information is maintained. In SCI, the directory is a distributed sharing list maintained by the processor caches themselves. For example, if processors A, B, and C are caching some location, then the cache entries storing this location will form a doubly-linked list. At main memory, only a pointer to the processor at the head of the linked list is maintained. In contrast, DASH places all the directory information with main memory.

The main advantage of the SCI scheme over DASH is that the amount of directory pointer storage grows naturally with the number of processors in the system. In DASH, the maximum number of processors must be fixed beforehand, or the system must support some form of limited directory information. On the other hand, the SCI directory memory would normally employ the same SRAM technology used by the processor caches while the DASH directory is implemented in main memory DRAM technology. Another feature of SCI is that it guarantees forward progress in all cases, including the pathological "live-lock" case alluded to in section 4.6.

The primary disadvantage of the SCI scheme is that the distribution of the individual directory entries increases the complexity and latency of the directory protocol, since additional directory update messages must be sent between processor caches. For example, on a write to a shared block cached by $N + 1$ processors (including the writing processor), the writer must perform the following actions: (i) detach itself from the sharing list; (ii) interrogate memory to determine the head of the sharing list; (iii) acquire head status from the current head; and (iv) serially purge the other processor caches by issuing invalidation requests and receiving replies indicating the next processor in the list. Altogether, this amounts to $2N + 8$ messages including N serial directory lookups. In contrast, DASH can locate all sharing processors in a single directory lookup and invalidation messages are serialized only by the network transmission rate. Likewise, many read misses in SCI require more inter-node communication. For example, if a block is currently cached, processing a read miss requires four messages since only the head can supply the cache block. Furthermore, if a miss is replacing a valid block in the processor's cache, the replaced block must be detached from its sharing list.

Recently, the SCI working committee has proposed a number of extensions to the base protocol that address some of these shortcomings. In particular, the committee has proposed additional directory pointers that allow sharing lists to become sharing trees, the support for request forwarding, and the use of a clean cached state. While these extensions reduce the differences between the two protocols, they also add complexity. The fundamental question is what set of features leads to better performance at a given complexity level. As in the design of other hardware systems, this requires a careful balance between optimizing the performance of common operations without adding undue complexity for uncommon ones. The lack of good statistics on scalable shared memory machines, however, makes the identification of the common cases difficult. Thus, a complete comparison of the protocols is likely to require actual implementations of both designs and much more experience with this class of machines.

9 Summary and Status

Distributed directory-based coherence protocols such as the DASH protocol allow for the scalability of shared-memory multiprocessors with coherent caches. The cost of scalability is the added complexity of directory based schemes compared with existing snoopy, bus-based coherence protocols. The complexity arises primarily from the lack of a single serialization point within the system and the lack of atomic operations. Additional complexity stems simply from the larger set of components that interact to execute the protocol and the deeper hierarchy within the memory system.

Minimizing memory latency is of paramount importance in scalable systems. Support for coherent caches is the first step in reducing latency, but the memory system must also be optimized towards this goal. The DASH protocol attempts to minimize latency through the use of the release consistency model, cache-to-cache transfers, a forwarding control strategy and special purpose operations such as prefetch and update write. Adding these latency reducing features must, of course, be traded off with the complexity needed to support them. All of the above features were added without a significant increase in the complexity of the hardware.

Verification of a complex distributed directory-based cache coherence protocol is a major challenge. We feel that verification through the use of test scripts and extensive random testing will provide an acceptable level of confidence. The design effort of the prototype is currently in the implementation phase. A functional simulator of the hardware is running as well as a gate level simulation of the directory card. We plan to have a 4 cluster, 16 processor system running during the summer of 1990. This prototype should serve as the ultimate verification of the design and provide a vehicle to fully evaluate the design concepts discussed in this paper.

10 Acknowledgments

The DASH project team is composed of a number of graduate students and faculty within the Computer System Laboratory at Stanford. Many related topics are being researched and the results of much of this work has influenced the design of the DASH architecture and coherence protocol. Besides the authors, a number of others have directly contributed to the development of DASH. In particular, we would like to thank Wolf-Dietrich Weber for creating the DASH simulator, Helen Davis and Stephen Goldschmidt for modifying their Tango simulator to interact with the DASH simulator, and Bruce Kleinman for developing the DASH protocol verifier. Likewise, we want to recognize research engineer Dave Nakahira who has made significant contributions to the design of the DASH hardware. We also wish to thank Valid Logic Systems who has donated the CAE software used to develop the DASH prototype.

This research is supported by DARPA contract N00014-87-K-0828. Dan Lenoski is supported by Tandem Computers Incorporated.

References

[1] A. Agarwal, R. Simoni, J. Hennessy, and M. Horowitz. An evaluation of directory schemes for cache coherence. In *Proc. of the 15th Annual Int. Sym. on Computer Architecture*, pages 280–289, June 1988.

[2] J. Archibald and J.-L. Baer. An economical solution to the cache coherence problem. In *Proc. of the 12th Int. Sym. on Computer Architecture*, pages 355–362, June 1985.

[3] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Trans. on Computer Systems*, 4(4):273–298, 1986.

[4] F. Baskett, T. Jermoluk, and D. Solomon. The 4D-MP graphics superworkstation: Computing + graphics = 40 MIPS + 40 MFLOPS and 100,000 lighted polygons per second. In *Proc. of the 33rd IEEE Computer Society Int. Conf. – COMPCON 88*, pages 468–471, February 1988.

[5] W. C. Brantley, K. P. McAuliffe, and J. Weiss. RP3 processor-memory element. In *Proc. of the 1985 Int. Conf. on Parallel Processing*, pages 782–789, 1985.

[6] L. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Trans. on Computers*, C-27(12):1112–1118, December 1978.

[7] W. J. Dally. Wire efficient VLSI multiprocessor communication networks. In *Stanford Conference on Advanced Research in VLSI*, 1987.

[8] M. Dubois, C. Scheurich, and F. Briggs. Memory access buffering in multiprocessors. In *Proc. of the 13th Annual Int. Sym. on Computer Architecture*, pages 434–442, June 1986.

[9] C. M. Flaig. VLSI mesh routing systems. Technical Report 5241:TR:87, California Institute of Technology, May 1987.

[10] K. Gharachorloo, D. Lenoski, J. Laudon, P. Gibbons, A. Gupta, and J. Hennessy. Memory consistency and event ordering in scalable shared-memory multiprocessors. In *Proc. of the 17th Annual Int. Sym. on Computer Architecture*, June 1990.

[11] S. R. Goldschmidt and H. Davis. Tango introduction and tutorial. Technical Report CSL-TR-90-410, Stanford University, January 1990.

[12] P1596 Working Group. P1596/Part IIIA - SCI Cache Coherence Overview. Technical Report Revision 0.33, IEEE Computer Society, November 1989.

[13] A. Gupta and W.-D. Weber. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. Technical Report CSL-TR-90-417, Stanford University, March 1990.

[14] B. Kleinman. *DASH Protocol Verification*, EE-391 Class Project Report, December 1989.

[15] T. Knight. Architectures for artificial intelligence. In *Int. Conf. on Computer Design*, 1987.

[16] L. Lamport. How to make a multiprocessor computer that correctly executes multiprocess programs. *IEEE Trans. on Computers*, C-28(9):241–248, September 1979.

[17] D. Lenoski, J. Laudon, K. Gharachorloo, A. Gupta, J. Hennessy, M. Horowitz, and M. Lam. Design of the Stanford DASH multiprocessor. Technical Report CSL-TR-89-403, Stanford University, December 1989.

[18] B. O'Krafka and A. R. Newton. An empirical evaluation of two memory-efficient directory methods. In *Proc. of the 17th Annual Int. Sym. on Computer Architecture*, June 1990.

[19] M. S. Papamarcos and J. H. Patel. A low overhead coherence solution for multiprocessors with private cache memories. In *Proc. of the 11th Annual Int. Sym. on Computer Architecture*, pages 348–354, June 1984.

[20] C. K. Tang. Cache design in the tightly coupled multiprocessor system. In *AFIPS Conf. Proc., National Computer Conf., NY, NY*, pages 749–753, June 1976.

[21] J. Willis. Cache coherence in systems with parallel communication channels & many processors. Technical Report TR-88-013, Philips Laboratories - Briarcliff, March 1988.

[22] D. A. Wood, G. A. Gibson, and R. H. Katz. Verifying a mulitprocessor cache controller using random case generation. Technical Report 89/490, University of California, Berkeley, 1988.

[23] W. C. Yen, D. W. Yen, and K.-S. Fu. Data coherence problem in a multicache system. *IEEE Trans. on Computers*, C-34(1):56–65, January 1985.

Appendix A: Coherence Transaction Details

```
if (Data held locally in shared state by processor or RAC)
   Other cache(s) supply data for fill;

else if (Data held locally in dirty state by processor or RAC) {
   Dirty cache supplies data for fill and goes to shared state;
   if (Memory Home is Local)
      Writeback Data to main memory;
   else
      RAC takes data in shared-dirty state;
   }

else if (Memory home is Local) {
   if (Directory entry state != Dirty-Remote)
      Memory supplies read data;
   else {
      Allocate RAC entry, mask arbitration and force retry;
      Forward Read Request to Dirty Cluster;
      PCPU on Dirty Cluster issues read request;
      Dirty cache supplies data and goes to shared state;
      DC sends shared data reply to local cluster;
      Local RC gets reply and unmasks processor arbitration;
      Upon local processor read, RC supplies data and the
         RAC entry goes to shared state;
      Directory entry state = Shared-Remote;
      }
   }

else /* Memory home is Remote */ {
   Allocate RAC entry, mask arbitration and force retry;
   Local DC sends read request to home cluster;
   if (Directory entry state != Dirty-Remote) {
      Directory entry state = Shared-Remote, update vector;
      Home DC sends reply to local RC;
      Local RC gets reply and unmasks processor arbitration;
   else {
      Home DC forwards Read Request to dirty cluster;
      PCPU on dirty cluster issues read request and DC sends
         reply to local cluster and sharing writeback to home;
      Local RC gets reply and unmasks processor arbitration;
      Home DC gets sharing writeback, writes back dirty data,
         Directory entry state = Shared-Remote, update vector;
      }
   Upon local processor read, RC supplies the data and the
      RAC entry goes to shared state;
   }
```

Figure 7: Normal flow of read request bus transaction.

```
if (Memory Home is Local) {
   Writeback data is written back into main memory;
   }
else /* Memory Home is Remote */ {
   Writeback request sent to home;
   Writeback data is written back into main memory;
   Directory entry state = Uncached-Remote, update vector;
   }
```

Figure 8: Normal flow of a write-back request bus transaction.

```
if (Data held locally in dirty state by processor or RAC)
   Dirty cache supplies Read-Exclusive fill data and
      invalidates self;

else if (Memory Home is Local)  {
   switch (Directory entry state) {

      case Uncached-Remote :
         Memory supplies data, any locally cached copies
            are invalidated;
         break;

      case Shared-Remote :
         RC allocates an entry in RAC with DC specified
            invalidate acknowledge count;
         Memory supplies data, any locally cached copies are
            invalidated;
         Local DC sends invalidate request to shared clusters;
         Dir. entry state = Uncached-Remote, update vector;
         Upon receipt of all acknowledges RC deallocates RAC
            entry;
         break;

      case Dirty-Remote :
         Allocate RAC entry, mask arbitration and force retry;
         Forward Read-Exclusive Request to dirty cluster;
         PCPU at dirty cluster issues Read-Ex request,
            Dirty cache supplies data and invalidates self;
         DC in dirty cluster sends reply to local RC;
         Local RC gets reply from dirty cluster and unmasks
            processor arbitration;
         Upon local processor re-Read-Ex, RC supplies data,
            RAC entry is deallocated and
            Dir. entry state = Uncached-Remote, update vector;
      }
   }
else /* Memory Home is Remote */ {
   RC allocates RAC entry, masks arbitration and forces retry;
   Local DC sends Read-Exclusive request to home;
   switch (Directory entry state) {

      case Uncached-Remote :
         Home memory supplies data, any locally cached copies
            are invalidated, Home DC sends reply to local RC;
         Directory entry state = Dirty-Remote, update vector;
         Local RC gets Read-Ex reply with zero invalidation
            count and unmasks processor for arbitration;
         Upon local processor re-Read-Ex, RC supplies data and
            RAC entry is deallocated;
         break;

      case Shared-Remote :
         Home memory supplies data, any locally cached copies
            are invalidated, Home DC sends reply to local RC;
         Home DC sends invalidation requests to sharing
            clusters;
         Directory entry state = Dirty-Remote, update vector;
         Local RC gets reply with data and invalidate acknow-
            ledge count and unmasks processor for arbitration;
         Upon local processor re-Read-Ex, RC supplies data;
         Upon receipt of all acknowledges RC deallocates RAC
            entry;
         break;

      case Dirty-Remote :
         Home DC forwards Read-Ex request to dirty cluster;
         PCPU at dirty cluster issues Read-Ex request,
            Dirty cache supplies data and invalidates self;
         DC in dirty cluster sends reply to local RC with
            acknowledge count of one and sends Dirty Transfer
            request to home;
         Local RC gets reply and acknowledge count and unmasks
            processor for arbitration;
         Upon local processor re-Read-Ex, RC supplies data;
         Upon receipt of Dirty Transfer request, Home DC
            sends acknowledgment to local RC,
           Home Dir. entry state = Dirty-Remote, update vector;
         Upon receipt of acknowledge RC deallocates RAC entry;
      }
   }
```

Figure 9: Normal flow of read-exclusive request bus transaction.