Implementing a Directory-Based Cache Consistency Protocol

Richard Simoni

Technical Report No. CSL-TR-90-423

March 1990

This research has been supported by DARPA contract N00014-87-K-0828.

Implementing a Directory-Based Cache Consistency Protocol

Richard Simoni

Technical Report: CSL-TR-90-423

March 1990

Computer Systems Laboratory
Departments of Electrical Engineering and Computer Science
Stanford University
Stanford, California 94305-4055

Abstract

Directory-based cache consistency protocols have the potential to allow shared-memory multiprocessors to scale to a large number of processors. While many variations of these coherence schemes exist in the literature, they have typically been described at a rather high level, making adequate evaluation difficult. This paper explores the implementation issues of directory-based coherency strategies by developing a design at the level of detail needed to write a memory system functional simulator with an accurate timing model.

The paper presents the design of both an invalidation coherency protocol and the associated directory/memory hardware. Support is added to prevent deadlock, handle subtle consistency situations, and implement a proper programming model of multiprocess execution. Extensions are delineated for realizing a *multiple-threaded* directory that can continue to process commands while waiting for a reply from a cache. The final hardware design is evaluated in the context of the number of parts required for implementation.

Key Words and Phrases: Directory-based cache consistency, cache coherency, scalable shared-memory multiprocessors.

Table of Contents

List of Figures

List of Tables

Implementing a Directory-Based Cache Consistency Protocol

Richard Simoni
Computer Systems Laboratory
Stanford University
Stanford, CA 94305

1. Introduction

Directory-based cache consistency schemes have the potential to scale shared-memory multiprocessors to a large number of processors [1, 4]. For this reason, we are interested in taking a closer look at the advantages and disadvantages of this class of protocols. But it is difficult to draw any certain conclusions about these schemes with only a vague notion of how they function and are implemented in a real system. The purpose of this paper is to clarify these notions by presenting a reasonably detailed design of the hardware needed to implement a directory-based cache consistency protocol.

We have chosen to approach the design at a level of detail that will allow a functional simulator with an accurate timing model to be written. Towards this end, our design shows every necessary piece of hardware in the datapath, but ignores controller details when it is apparent how one would go about designing the controller. In addition, we formally define the consistency protocol by identifying the states and their corresponding actions, including both messages that must be sent and state changes within the directory. The resulting "action table" could then define a simulator's functionality, while the detailed hardware specification can be used to derive an accurate timing model.

Because we would like the reader to understand our design decisions, we have chosen to not simply present the final design, but rather step the reader through the design process. We start in the next section by stating the assumptions we make about the system in which our protocol will operate. We then enumerate the types of messages required to implement consistency, and identify the conditions that must be satisfied to avoid deadlock. These conditions help us design a strategy for communication between the hardware units within a node. Once this is done, we know enough about how messages are handled in a node to further refine the message definitions to handle several subtle cases correctly. We then add the necessary support for a model of parallel execution, such as ***strong ordering*** or ***weak ordering.*** Once the protocol definition is complete, we design the directory hardware itself, including a precise definition of the contents of a directory entry, the hardware necessary to implement the directory and the messages required by the protocol, and a state transition table for the directory controller. Finally, we examine the size of the resulting hardware.

This is a revised edition of an unpublished report of July, 1988.

2. System Assumptions

In order to have a starting point for our design, we must make some assumptions about the system for which the directory-based consistency scheme is intended. First and foremost, we assume the system is a shared-memory multiprocessor that uses a large number of high-performance CPUs. We assume that each processor is paired with its own cache. And finally, we assume that main memory is distributed across the processors. That is, part of the globally-shared main memory is physically local to each processor/cache pair. Therefore, the minimum latency to memory for a processor request is lower if the request is to a location that happens to be local than if the request is to a remote location. This system configuration is shown in Figure 2-l. We further assume the directory bits for each chunk of main memory are located with that chunk, i.e., the directory bits are distributed across the processor nodes along with main memory.

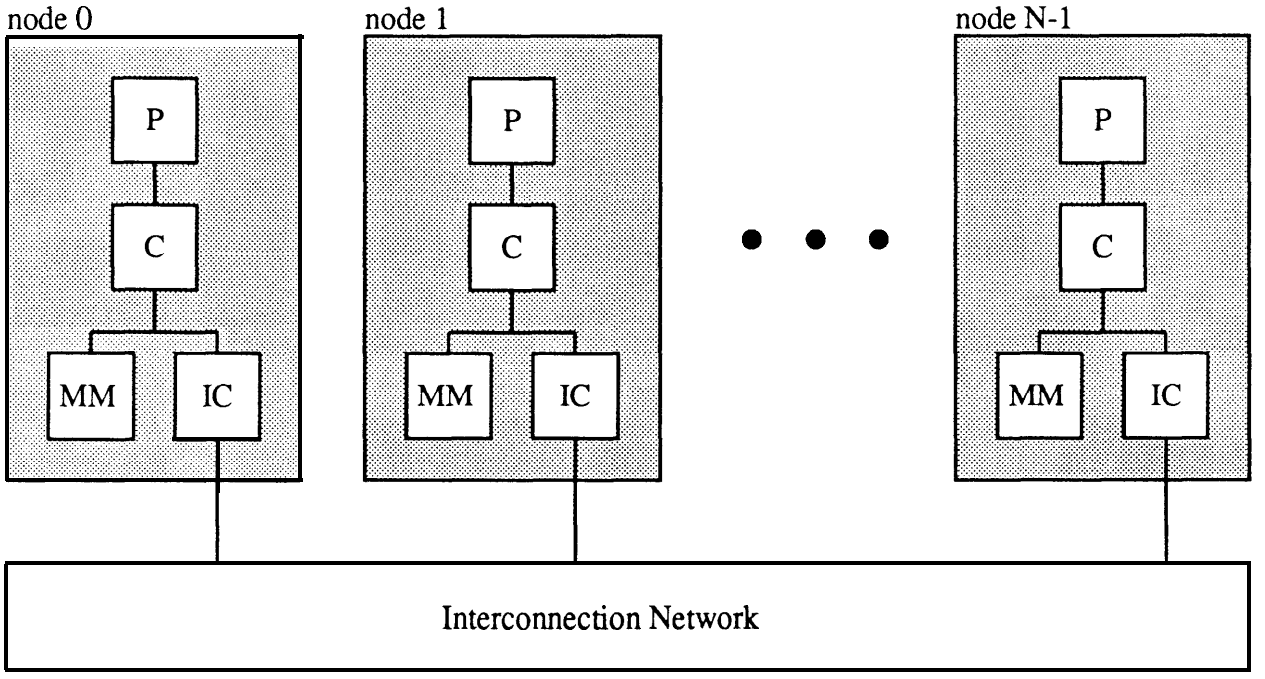

Figure 2-l: The basic system. Each node consists of a processor (P), cache (C), main memory (MM), and interconnect controller (IC).

A wide variety of networks satisfy our assumptions about the interconnect. We assume the network has only point-to-point capability, i.e., there is no hardware mechanism to implement broadcasts efficiently. Each message on the network is sent with a source and destination processor node number. We assume that messages between any two given nodes are delivered in the same order that they were sent. The interface at each node to the network is called the ***interconnect controller*** (IC). We assume it provides its node with reliable communication to and from the other nodes in the system.

3. Command Messages

The basic mechanism by which consistency is maintained in a directory-based protocol is the passing of commands in messages between the caches and the directories associated with main memory. A typical sequence

begins when the cache cannot satisfy a processor request without communicating with memory; an example is a cache miss. In this case the cache will send a command to a directory instructing it to supply the missing block. The directory may be able to do this directly, or it may need to take other action before responding, such as sending a command to another cache directing it to return its dirty copy of the block

Thus, we have two varieties of messages. The first type is cache-to-main-memory (abbreviated C→MM) commands, which are initiated by a cache any time it cannot immediately satisfy a processor's request. The second type is main-memory-to-cache (abbreviated MM→C) commands, which are issued by a directory to instruct caches to take some action before it can respond to a C→MM command it has received.

A quick perusal of the possible cache events that can occur [1] shows that we need four different C→MM commands:

- ***readlnon-exclusive (readlnon-ex).*** The cache issues this command on a read miss in order to get a copy of the block from main memory.
- *read/exclusive* ***(readlex).*** In the case of a write miss, the cache uses this command to get an exclusive copy of the block from main memory.
- ***exclusive (ex).*** When the cache encounters a write hit on a clean block, it must ask the directory for exclusive access to this block; it will then be the single cache with the dirty copy of the block.
- ***writeback.*** This command is used by the cache to write back a dirty block to main memory if, for example, it is replacing the block with another one.

As previously mentioned, a directory may need to have other caches take action when one of these C→MM commands is received. The three MM→C commands are as follows:

- ***copyback.*** This command tells the target cache to copyback the indicated block to main memory. The block need not be invalidated, however, This command is required when there is a read miss in a cache on a block that is dirty in another cache.
- *flush.* A cache receiving this command should copy the specified block back to main memory and invalidate the block. This case occurs when a cache has a dirty copy of the block and there is a write miss on the block in another cache.
- ***invalidate.*** This command should cause the receiving cache to invalidate the indicated block It is issued when another cache requests exclusive access to a block, either through a *read/ex* or ex C→MM command

These C→MM and MM→C commands are summarized in Table 3-l. Note that many of the commands require a ***reply,*** e.g., to return a block of data. We will modify and refine these commands and precisely define their replies later.

4. Avoiding Deadlock

There are several conditions that must be satisfied in the system to avoid the possibility of deadlock. A typical situation we must take care to avoid is one in which two nodes simultaneously send a message to each other and cannot continue until they receive a reply; neither node can reply because each is stuck waiting for a reply that will never come.

The key to avoiding these situations is to remember the worst-case sequence of transactions that can occur on a given block: a cache issues a C→MM command, and the directory has to issue multiple MM-X commands (to

C→MM commands
read/ex
read/non-ex
ex
writeback
MM→C commands
copyback
flush
invalidate

Table 3-1: Initial version of command messages.

caches other than the one that initiated the sequence) as a result; the target caches may then need to send replies back to the directory, which finally responds to the cache that issued the original request. We must ensure that there is no possibility of deadlock at each of the resources used in this worse-case scenario. In particular, we must make sure no messages become stuck (1) at a cache, (2) at a directory, and (3) in transit between a cache and a directory. We will examine each of these cases independently.

4.1. Cache Deadlock

Notice that when a cache receives a MM→C command (i.e., ***copyback, flush***, or ***invalidate),*** the cache never needs to issue another command to complete the requested action. That is, the actions required to carry out a MM→C command are self-contained in the destination cache. So the conditions sufficient to prevent deadlock at a cache degenemte to those necessary to ensure the cache eventually begins processing all commands it receives. In practice, there are only two reasons the cache might delay processing a command. First, if the CPU is accessing the cache, then the command must wait; however, the CPU will eventually complete its access, so this delay will not cause deadlock. Second, the cache may have issued an unrelated C→MM command command and awaits a reply. This situation is more dangerous.

Consider an example, shown in Figure 4-l. There are two nodes, node 1 and node 2, and two data blocks, block A and block B. The directory information for blocks A and B resides in nodes 1 and 2, respectively. Block B is dirty in cache 1, and block A is dirty in cache 2. Now suppose processor 1 performs a read to block A, and at the same time, processor 2 performs a read to block B. Each cache sends an ***read/non-exclusive*** C→MM command to the appropriate directory and waits for the directory to return the data in a reply. Each ***directory*** issues a *copyback* MM→C command to the other node. But if these caches must put off the *copyback* until they receive the reply to ***the read/non-exclusive*** commands they sent, then they will wait forever, because neither reply can be returned by the directories until the *copyback* commands have been completed.

We can therefore state the condition that must be satisfied to prevent deadlock in the caches:

- A cache controller cannot wait to process and respond to a MM→C command until after it has received a reply from a pending C→MM request that it has issued.

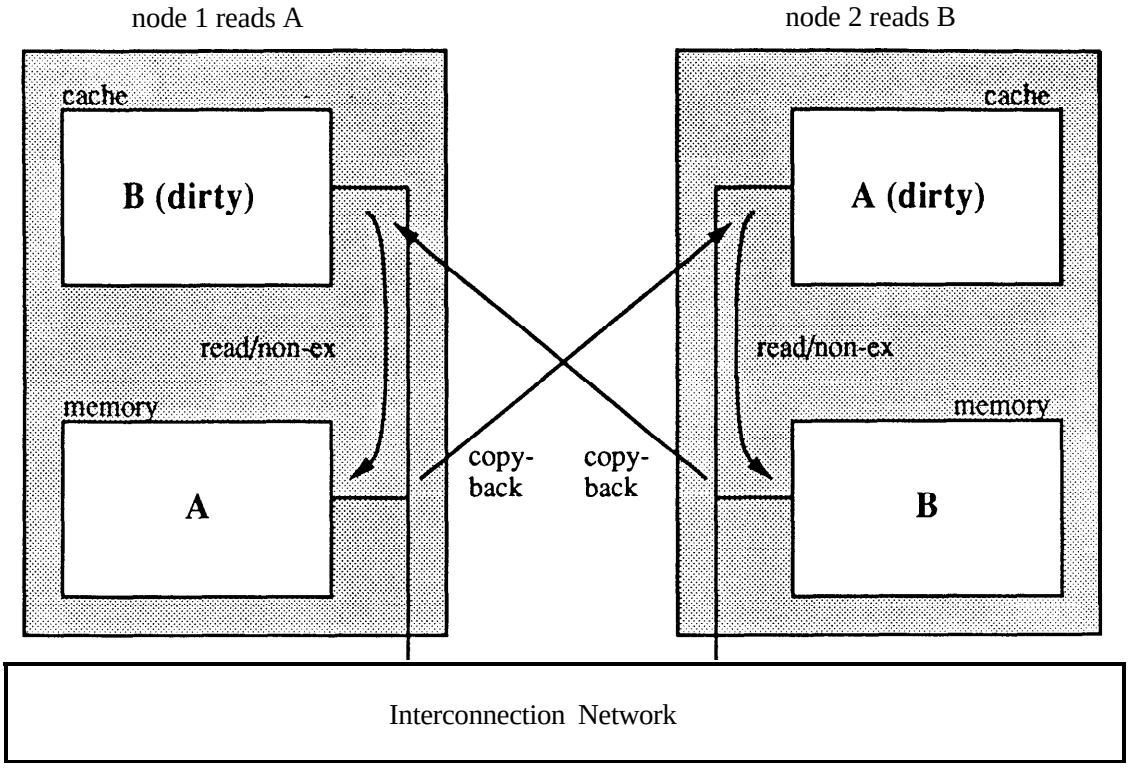

Figure 4-1: Deadlock situation that demonstrates why a cache must not wait for a reply to the C→MM command it has sent before processing received MM→C commands.

4.2. Directory Deadlock

Referring back to the the worst-case sequence of transactions, there are two distinct processing phases a directory must deal with to fully complete a request. First, the directory must handle the C→MM command request, which may include sending MM→C commands to caches. Second, the directory must process the replies from those caches, which may include sending a reply to the cache that sent the original C→MM command. Once access to the directory is gained, both of these processing phases are completed without blocking at the directory. We therefore need only ensure that all C→MM commands and replies to MM→C commands attain directory access.

Commands and replies from caches compete solely with each other for the directory resource. Replies from caches can always be handled immediately; however, there can be constraints on the C→MM commands. For instance, a command for an address cannot be executed in the directory if a transaction is already pending for that address; otherwise, cache consistency will be compromised. The directory may therefore block on some C→MM commands. It must be ensured that this blocking does not prevent replies to MM→C commands from accessing the directory, since these replies are required to release the blocked command. We can state this condition as follows:

- A directory controller cannot delay processing replies to a MM→C command until after processing a C→MM command command that cannot be immediately processed.

4.3. Transit Deadlock

Now that we have specified conditions sufficient to prevent deadlock at the cache and the directory, we need only assure that messages between them do not deadlock in transit. The route taken by a message includes both the channels of the interconnection network between nodes and the intra-node communication paths. Since the messages in the system are the only users of these communication resources, we can prevent deadlock by guaranteeing that no messages permanently block other messages. The network, for instance, must not be built so that a C→MM command holds the link between the cache and directory if this may prevent a resulting MM→C command to a different cache from reaching its destination. In our design we assume that all replies to a command require a separate transaction on the interconnection network. Not only is this assumption safe in terms of avoiding deadlock, but also consumes less network bandwidth since the network is not tied up needlessly during command processing. [1]

Within a node, the directory controller should either not see arriving MM→C commands at all, or should be able to forward them to the cache regardless of the current state of the controller. Otherwise, deadlock may occur if each directory on two nodes sends a MM→C command to the other node, for the commands may be blocked from the destination cache by the local directory. We can generalize the above concepts into a condition sufficient to prevent transit deadlock:

- The interconnection network must be non-blocking, and each hardware unit within a node cannot block messages intended for another unit in the same node.

5. Traffic Cop Problem

An important problem to solve is how to efficiently pass data between the major subsystems within each node. For our purposes we will consider three subsystems. The ***directory controller (DC)*** maintains the contents of the directory bits. The ***interconnect controller (IC)*** implements the interface between the node and the interconnection network. For our design we can view it abstractly as a mechanism that reliably sends and receives C→MM and MM-42 commands. The ***cache controller*** (CC) maintains the internal cache directory (comprised of tags, valid bits, dirty bits, etc.) and satisfies memory requests from the processor.

Figure 5-l shows the communication paths that must be supported within a node to transfer commands (and replies) between the controllers. The cache controller issues C→MM commands, which may be directed to either the local directory or to a remote directory via the interconnect controller. Similarly, the MM→C commands from the directory controller may go to the local cache or to remote caches. Finally, commands arriving at the node through the interconnect controller may go to the cache or the directory, depending on the command type (C→MM or MM→C). We call the problem of directing each of these messages to its destination efficiently ***the traffic cop problem.***

[1] **However, this assumption does result in a longer command latency since arbitration for the interconnect link must occur twice, once for the command and once for the reply. However, this additional arbitration &lay will be small compared to the total latency, and should therefore have negligible effect on the system performance.**

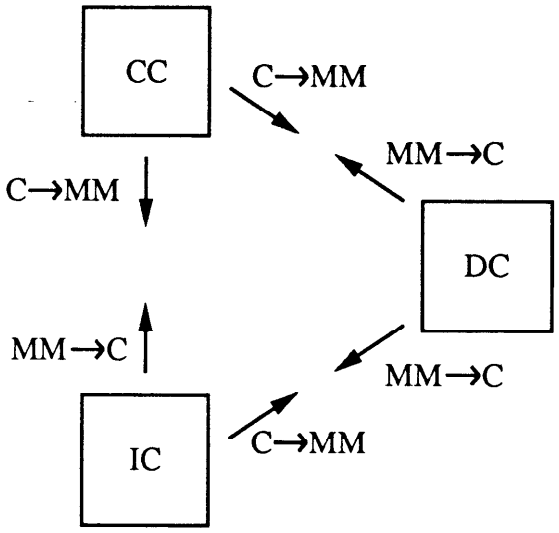

Figure 5-l: Overview of the traffic cop problem.

5.1. Message Queueing

Because we do not want to halt a producing resource just because the consumer happens to be momentarily busy, we begin our design by placing a FIFO queue at the input to each of the controllers. This has additional benefits, as we will see shortly. Also notice that some sort of arbitration is needed on the input to these queues, since each queue is fed by two sources. For example, the directory controller must accept C→MM commands from both the local cache and the interconnect controller. The resulting system is shown in Figure 5-2.

Let us further consider the arbitration on the input to the cache and the directory. Say the cache controller in a node prepares a command for the directory in the same node. It then gains access through the arbiter to the directory input queue since the interconnect controller currently has no commands for the directory. But as soon as it begins to transfer data into the queue, a C→MM command arrives at the interconnect controller. Now the interconnect controller cannot transfer the command to the directory input queue until the cache has completed moving its own command into the queue. There are several design options for the interconnect controller when this situation is encountered. The interconnect controller could abort the message, causing it to be sent again later. This is obviously not an attractive alternative since additional network traffic results from an operation that should be fairly innocuous, i.e., the transfer of data from a cache to the directory within a node. Another option is to buffer the incoming message locally in the interconnect controller, perhaps in its own FIFO queue. While this seems to be a reasonable design choice, it does add delay as well as some hardware complexity since incoming messages should be buffered in the interconnect controller only if the desired queue input is busy.

Our proposed design solves the problem as follows. Rather than try to implement buffering in the interconnect controller, we provide a separate input queue at the cache and directory for each of the input sources. This arrangement is shown in Figure 5-3. Note that the arbiters on the cache and directory queue inputs are eliminated, so commands to these controllers can be latched into the input queue immediately. Of course, the arbitration problem has really just been moved from the input of each queue to the output. But the problem is much simpler there, since neither of the arbitrating parties require “instant” processing as the interconnect controller does when an incoming command arrives. At the output, the consumer simply chooses to pick the next command from one queue or the other based on the queue empty/full information.

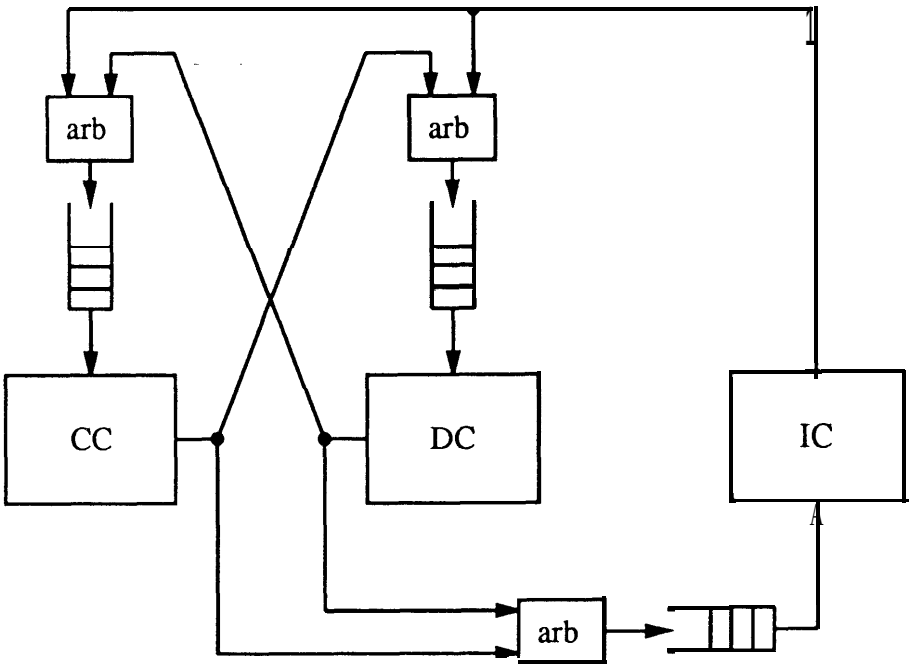

Figure 5-2: A solution to the traffic cop problem.

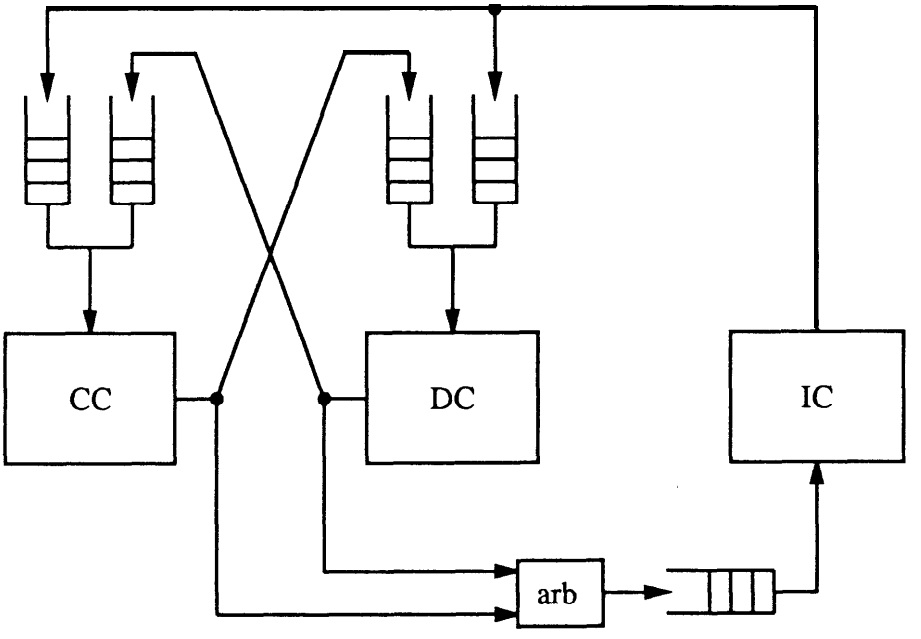

Figure 5-3: A modified configuration that replaces the arbiters on the inputs to the cache and directory controllers with separate input queues.

Note that we have left the arbiter for the interconnect controller input queue in the design, rather than replace it with two queues as well. This is because the two sources driving the arbiter, namely the cache and directory controllers, should be able to wait for the queue to become free without significant additional hardware complexity

or performance penalty. If it is found that this is not the case, then the double-queue solution can be used here as well.

Another benefit comes from providing the cache and directory with separate queues to accept incoming messages from the interconnect controller. A likely implementation technology for the queues are the catalog FIFO memories based on fast static RAM technology. One feature of these parts is that there are separate data pins for the queue input and output, and the input and output ends of the queue can be controlled asynchronously with respect to each other. We can use this feature to our advantage if the interconnect in the system is asynchronous relative to the internal node, as is often the case. The FIFO memories can then provide the required asynchronous interface between the interconnect and the node. In other words, the interconnect controller would gate the data from the network directly into the directory and cache input queues rather than an internal buffer. Figure 5-4 shows the design again, highlighting the sections that are asynchronous relative to the processor clock.

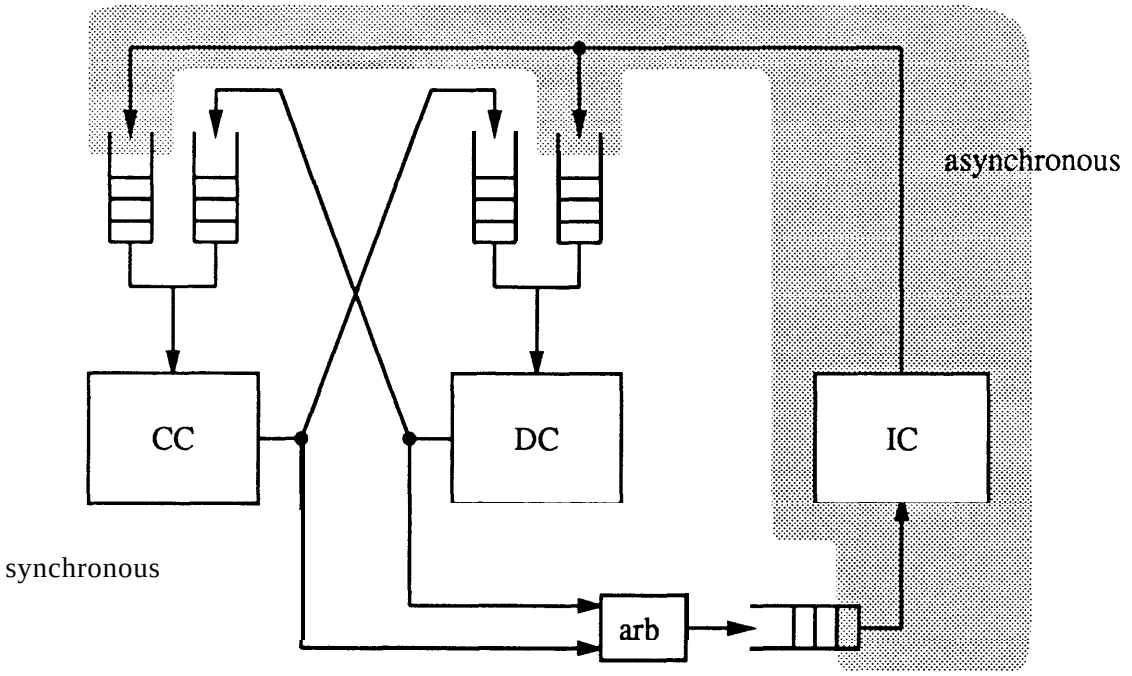

Figure 5-4: Highlighting the parts of the system with asynchronous signals.

Unfortunately, the solution to the traffic cop problem shown in Figure 5-3 has a deadlock problem. Recall the section about preventing directory deadlock (Section 4.2). The condition for avoiding deadlock is that the directory cannot block replies from caches simply because a C→MM command is blocked. As we noted, the directory cannot proceed with a command if there is already another transaction pending for the same address. If this happens, the logical action for the directory is to temporarily stop pulling commands from the queue, i.e., block the queue. But if the queue is blocked, then the directory can receive no replies and the pending transaction that prevents the directory from proceeding will never complete. Blocking the queue violates the condition necessary to prevent directory deadlock.

In order to allow reply messages to bypass the blocked queue, we will provide two special ***reply queues*** on the input to the directory controller where replies are routed. Our revised (and final) solution to the traffic cop problem includes these extra queues for the directory controller as well as the original two ***command queues*** and is shown in

Figure 5-5. The mechanism works by encoding a ***reply bit*** in each message that indicates whether it is a command or a reply to a previous command. When a message arrives for the directory controller from the cache or interconnect controllers, it is gatedinto the corresponding command or reply queue depending on the value of the reply bit.

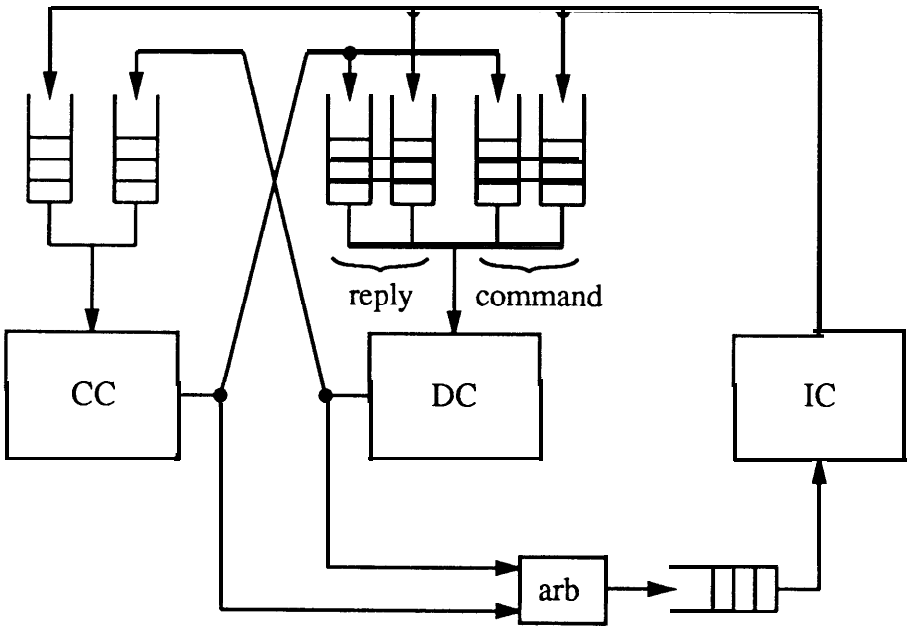

Figure **5-5:** Our final solution to the traffic cop problem includes separate queues on the input to the directory controller for replies and commands.

Finally, we must decide what should be done if a message arrives at a node for a queue that is full. One option is to let the message back up in the network by holding the communication link until the queue can accept input. This is feasible only if the network continues to allow other messages to get through to the node; otherwise, we have violated the condition for avoiding transit deadlock (see Section 4.3). An alternative strategy is aborting the message, allowing it to be sent again later. Since the queue will become full infrequently, the mechanism we choose to handle this case has negligible performance implications. We should implement whatever option is easiest, given the particular interconnection network in the system.

5.2. Multiple Threads

An issue we have not considered up to ***this*** point is ***single-threaded*** versus ***multiple-threaded*** directory controllers. A single-threaded controller can have the transactions for no more than one C→MM request pending at a given time. This restriction is not placed on multiple-threaded controllers. The advantage of multiple threads is that the directory can proceed with other C→MM requests while waiting for caches to respond to MM→C commands the directory issued for a different block. Although our design assumes a single-threaded directory controller, Section 8.5 will discuss the additional protocol and hardware support needed to implement multiple threads.

Let us again consider the reply queues of Figure 5-5 in the context of multiple threads. At first glance it appears that the reply queues are not necessary to prevent deadlock, because the directory does not stop processing commands from its input queue while it awaits a reply; therefore, a reply in the queue should eventually reach the

directory. Unfortunately, this argument is not entirely correct. While a multiple-threaded directory need not block its input queue whenever any transactions are pending (as does its single-threaded counterpart), it still must block the queue if any transactions *for the same address* as a new C→MM command are pending. Hence, deadlock is still possible, since a reply pertaining to a given address may get stuck in the queue behind an unrelated C→MM command for the same address.

As before, we can solve the deadlock problem by adding the reply queues to the multiple-threaded directory, and that may be the most desirable solution. Relative to the analogous case in a single-threaded controller, however, the directory input queue will block far less frequently. Perhaps commands and replies could use the same queue by having the directory discard the offending C→MM command rather than block it. The directory would then send a message to the slighted cache instructing it to retry its command. If this situation occurs infrequently then there should not be a noticeable effect on performance; furthermore, less hardware is required since we no longer need the reply queues.

5.3. Routing

We have now defined the buses used to move messages between the subsystems in each node. Each subsystem receives its messages by gating them into its input queue from the bus. Message header bits are decoded to drive the control inputs of these queues. Queues receiving messages from the interconnect controller use a bit indicating whether the message is intended for the cache or the directory, and a bit that is set if the message is a reply (the aforementioned ***reply bit).*** Messages originating in the cache controller or directory controller may be intended for the local node or a remote node. The processor number from the message header must therefore be compared with the local processor number[2]; if they match, the local directory or cache queue latches the message data. Otherwise, the message is intended for a remote node, and the interconnect controller must enqueue the message.

5.4. What Causes the Traffic Cop Problem?

As an aside, we note that the traffic cop problem is a manifestation of the system assumptions behind our design. Because the directory controller and cache controller are grouped together in a node, there are three resources per node (the cache controller, directory controller, and interconnect controller) that must communicate with each other. This implies that each controller must accept input from two sources; this is the cause of the traffic cop problem. Consider a system in which the memory and directory are not distributed across the processor nodes but instead communicate with all caches via the interconnection network. In such a system there are only two resources per node that need communicate with each other; the solution to the resulting traffic cop problem is much simpler and is shown in Figure 5-6. The simplicity in this system comes from the fact that no arbitration between two resources need occur within a node: all arbitration is handled at the interconnection network

6. Command Definition

Earlier, we defined the commands necessary to implement directory-based cache consistency (see Table 3-l). This section will refine the defmition of the commands to include the possible replies, to handle several subtle cases that occur, and to allow the proper implementation of a programming model of parallel execution.

---

[2]The local processor number can either be held in a software-writable register or set by DIP switches.

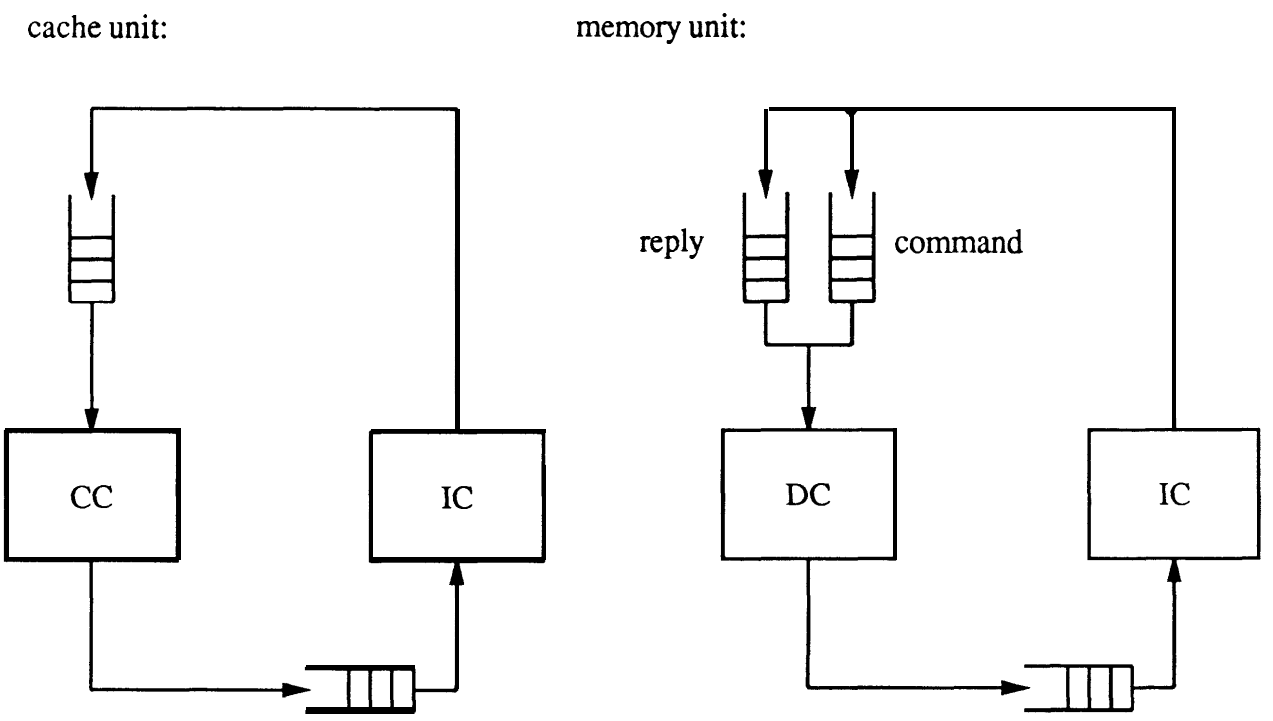

Figure 5-6: The solution to the traffic cop problem is simpler if the system has only two resources that must communicate on each node.

6.1. Replies

In Table 6-l we have modified the earlier table of commands to also show the possible replies to each command. The two C→MM read commands, *read/exclusive* and *read/non-exclusive,* expect a block of data back from memory, so the appropriate reply is *return data (retdata).* The exclusive command is issued by a cache that already has a clean copy of the block and needs to inform the directory that the block is about to be written. No data is returned in this case, so there is no reply to this command. Since the cache can proceed immediately after a *writeback* command, there is no need for a reply to be sent.

When a directory issues the MM→C *copyback* or *flush* commands, it expects to receive a block of data back from the cache; this *data* is carried in the *copyback data (cbdata)* reply. Since no data need be returned by the cache when it receives *an invalidate* command, no reply is necessary.

6.2. Subtle Cases

We must further refine our commands and replies to handle several less obvious cases. The first of these can occur when a block sits clean in several caches, and two processors try to write the block at the same time. Both caches proceed by issuing an *exclusive* command to the directory in charge of the block. It is clear that we only want one of the writes to proceed without intervention, i.e., the write corresponding to the first *exclusive* command to reach the directory. So we must not allow a write to a clean block to proceed in a cache until an acknowledge reply to the *exclusive* command has been received from the directory. We will call this reply *exclusive acknowledge (exuck).*

C→MM commands	Possible Replies
read/ex	retdata
read/non-ex	retdata
ex	(no reply)
writeback	(no reply)
MM→C commands	**Possible Replies**
copyback	cbdata
flush	cbdata
invalidate	(no reply)

Table 6-1: Command messages and their replies.

We now know what happens in the first cache to reach the directory with its ***exclusive*** command. It will receive ***an exclusive acknowledge*** reply, and proceed with the write. But what happens in the other cache that also issued an ***exclusive*** command? Before the directory gets around to processing the second cache's ***exclusive*** command, it will have sent ***this*** cache ***an invalidate*** command on the block, since the first cache was granted exclusive access. Now when the directory finally gets around to processing the second ***exclusive*** command, it must recognize from the dirty bit saved in the directory that the block now resides dirty in another cache. The appropriate action for the directory to take is to get the block back from the first cache (with a ***flush*** command) and to return the data to the second cache with a ***return data*** reply, as if the second cache had issued a ***read/exclusive*** instead of an ***exclusive*** command. This sequence of events is shown in Figure 6-l. So from a cache's point of view, whenever it issues an ***exclusive*** command, it may receive either ***an exclusive acknowledge*** reply or a ***return data*** reply. If it receives a ***return data*** reply, it is because the cache no longer contains the block due ***to an invulidute*** command it received between the time it issued the ***exclusive*** command and received a reply.

The second subtle case we must handle correctly is as follows. Say a cache issues a read command (***read/exclusive*** or ***read/non-exclusive)*** on a block that is dirty in another cache. This read command is put in the input command queue for the appropriate directory. Now the cache that currently has the block dirty decides to replace the block to make room for another by issuing a ***writebuck*** command, which is also put in the directory's input command queue. Eventually the directory processes the read command from the input queue. It sees that the line is dirty in a cache, and ***so*** it sends a ***flush*** or a ***copyback*** command to that cache. The directory is now deadlocked, for it is waiting for a cache to reply with a block of data that the cache can no longer supply since it has written the data back already. The block of data that the directory needs is stuck in the input queue. This situation is shown in Figure 6-2.

There are several approaches to solving this problem. The most obvious ***is*** to code all C→MM ***writebuck*** commands as replies (even though they are not sent in response to another message) by setting the reply bit in the command encoding. This method causes all writebacks to enter the reply queue at the input to the directory instead of the command queue. ***Thus,*** when the ***directory*** issues ***the copyback*** or ***flush*** command, the block of data will be in the correct place (the reply queue) even if the writeback was initiated by the cache before it receives the ***copyback*** or ***flush.***

This approach to solving the problem has some drawbacks. Because all ***writebucks*** now enter the reply queue, whenever the directory is awaiting a reply to any command it must be prepared to "wade through" and process any

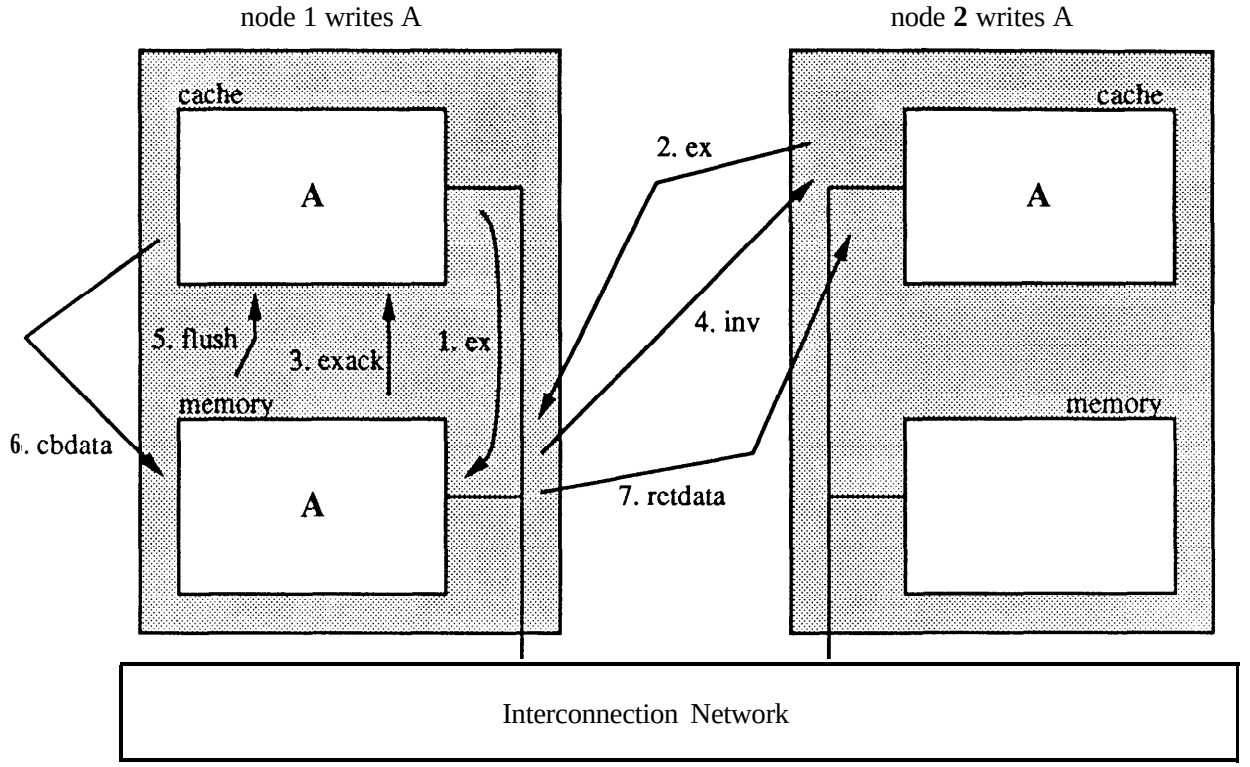

Figure 6-1: The situation that occurs when two processors write a block at the same time that they both have cached. Here, node 1 gains exclusive access first. Node 2 eventually receives a ***return data*** reply instead of ***the usual exclusive acknowledge*** reply.

writebucks that were queued prior ***to the true*** reply. ***These writebucks*** will be typically be blocks other than the block the directory is currently processing. The implication is that the directory controller will not only need multiple address registers, but will also have to delay addressing main memory until it has received the reply it is looking for, since the memory needs to be available until ***that time to handle writebucks.*** A multiple-threaded directory must already satisfy these conditions, making this an attractive solution for those systems.

For our single-threaded design, we choose a solution that uses a different strategy in the cache. We simply adopt the policy that when a cache initiates a ***writebuck*** command, it must not invalidate that line in its cache until it has received an acknowledgement from the directory. We call this acknowledgement a ***writebuck acknowledge*** *(wback)* reply. Now if the directory issues a *flush* or *copyback* command for a block that has already been written back and is stuck in the command queue, the cache can still supply the line in a reply to the directory.

There are two disadvantages to this technique. First, network traffic is increased for the case when a cache initiates a writeback. Hopefully, the caches in the system should be large enough so that cache-initiated writebacks, which are caused by interference in finite-sized caches, are relatively infrequent. Second, the cache issuing a ***writebuck*** cannot proceed with its replacement operation until it ***has*** received the ***writebuck acknowledge*** from the directory. However, there is nothing preventing the cache from sending the C→MM command to request the replacement block; in most ***cases we*** expect ***the writebuck acknowledge*** to ***arrive*** at ***the*** cache before the new data anyway.[3] Therefore, this solution should not significantly increase the average amount of time it takes a cache to

[3]The cache controller must buffer the new data if it returns before the *writeback acknowledge*, however. It is insufficient to simply leave the data in the cache input queue because this will block the *writeback acknowledge* from reaching the cache.

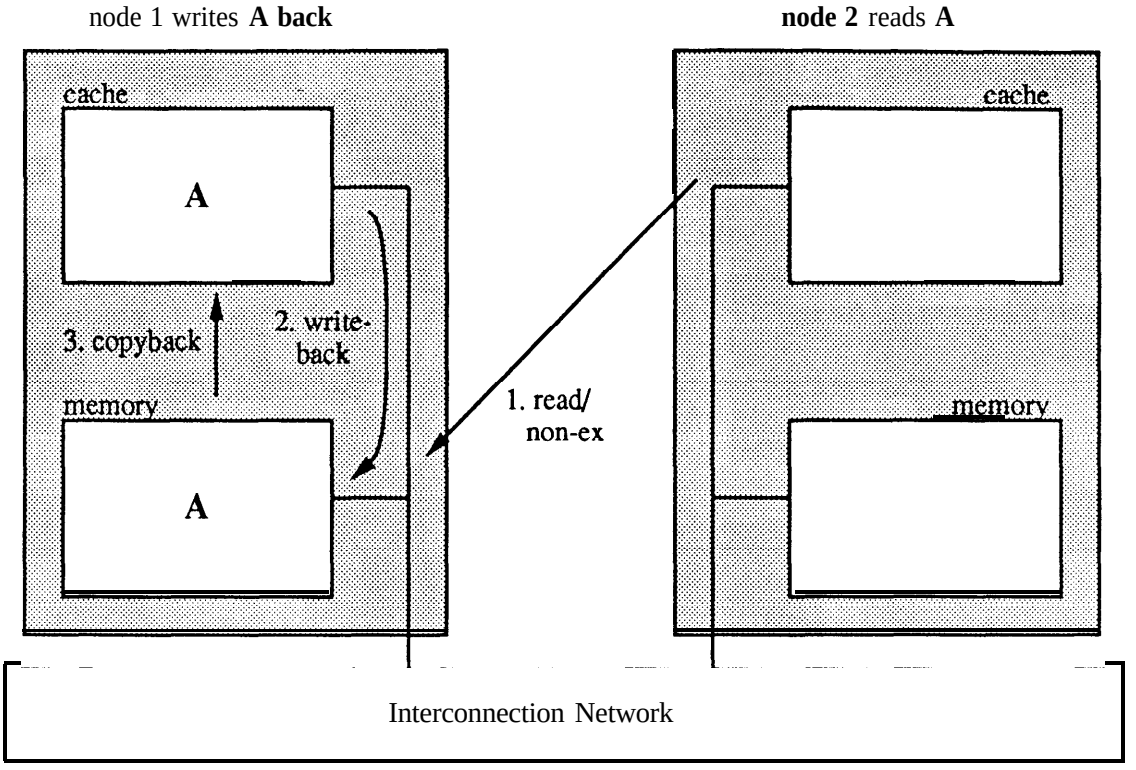

Figure 6-2: Another tricky situation. Node 2 reads a block that is dirty in node l's cache. But before the directory ***can*** issue a *copyback* command to the cache, the cache writes the block back ***to*** memory. Now when cache 1 receives the *copyback* command, it no longer has the block; the block is stuck in the command queue at the input to the directory.

replace a dirty block.

The only remaining issue is the ***writebuck*** command that can get stuck in the queue. If the directory sends a *copyback* or ***flush*** command and receives the block directly from the cache in a reply message, then the previously-issued ***writebuck*** command in the queue is no longer needed. So what should happen when the directory eventually services that command when it reaches the head of the queue? It is obvious that ignoring the command (i.e., taking no action) results in correct operation. But this strategy will require some special hardware to distinguish the ***writebuck*** commands that should be ignored from those that should be processed by the directory. It would be easier to simply always execute ***writebuck*** commands in the directory, since the situation in which they can be ignored is infrequent. Unfortunately, this results in incorrect operation, as demonstrated by an example. There are three caches involved, A, B, and C, and cache A contains a dirty block. Cache B has a write miss on the block, resulting in a ***read/exclusive*** command to be queued at the directory. Cache C has a read miss on the block; the directory input queue receives a *read/non-exclusive* command. At this point cache **A needs** to replace the block, so it sends a ***writebuck*** command to the directory. Figure 6-3 shows the state of the directory input queue at this point. Here is the sequence of events that occur as the directory processes each of the commands that is queued:

- *read/exclusive from B*. The directory sends a MM→C *flush* command to cache A. Cache A, which has not yet invalidated its copy of the block since it has not received a ***writebuck acknowledge,*** **returns the** block to the directory in a *copyback data* reply. The directory in turn supplies the block to cache B in a ***return data*** reply. Cache B then completes the write that caused the write miss. At this point the block is dirty in cache B.

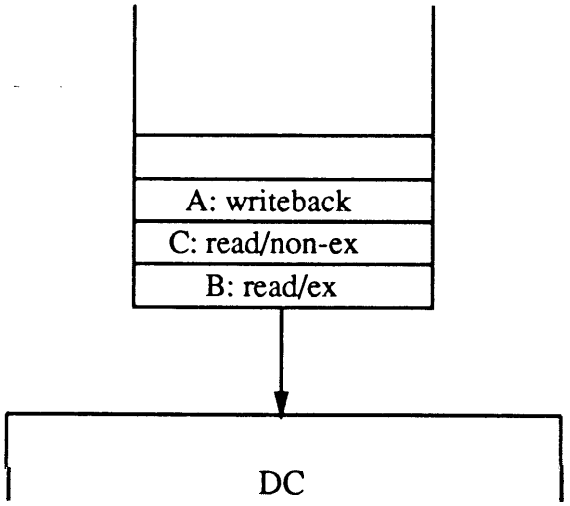

Figure 6-3: A series of commands in the queue that demonstrate why the ***writebuck*** command must be discarded.

- ***read/non-exclusive from C.*** The directory sends a MM→C *flush* command to cache B. Cache B returns the block to ***the*** directory ***in*** a *copyback data* reply. ***The*** directory sends the block to cache C in a ***return*** *data* reply and writes the block into main memory. At this point the block is clean and main memory will supply the block to any caches that miss on it.
- ***writebuck from A.*** This is the so-called "stale" writeback that was stuck in the queue when the directory needed the block. The data accompanying this command does not include the changes made in cache B when the write was completed. This ***writebuck*** command must be ignored by the directory; otherwise, main memory will supply stale data to the next cache that misses on the block.

All of the information needed to determine whether a given ***writebuck*** command should be discarded exists in the directory bits. ***Any*** cache performing a ***writebuck*** should have the block in a dirty state. When the directory controller receives the command, it should verify with the directory information that the processor that issued the ***writebuck*** is indeed supposed to have the block, and the block is indeed supposed to be dirty. If either of these conditions does not match, then the command should be ignored, since the data may be stale.

In another approach to solving the writeback problem of Figure 6-2, a cache would simply notify the directory if the cache receives a *copyback* or *flush* command for a block it does not contain. The directory would then reply to the cache that requested the ***data, instructing*** it to resend its *read/non-exclusive* or *read/exclusive* command. When the resent command reaches the directory, the writeback command will have completed and the *retdata* reply can be sent, assuming no other cache has claimed an exclusive copy by then. Although in general we prefer for efficiency reasons protocols that do not rely on resending messages, this particular solution is promising since we expect the situation depicted in Figure 6-2 to occur infrequently.

In order to handle the subtle cases presented in this section, we have added several replies to our command definition. Our current list of commands and possible replies to each command is shown in Table 6-2. In order to resolve the case where multiple caches request exclusive access to a block simultaneously, we have added two possible replies to the ***exclusive*** command: ***exclusive acknowledge*** and ***return*** *data*. And to allow directories to access blocks that have already been sent back ***in*** a ***writebuck*** command but are stuck in the input queue, we now require all caches ***to*** wait for a ***writebuck acknowledge*** reply before invalidating a block after replacing it via a ***writebuck*** command.

C→MM commands	Possible Replies
read/ex	retdata
read/non-ex	retdata
ex	exack retdata
writeback	wback
MM→C commands	**Possible Replies**
copyback	cbdata
flush	cbdata
invalidate	(no reply)

Table 6-2: Command messages revised to handle subtle cases.

7. Model of Parallel Execution

In any multiprocessor with cacheable shared data, an effort must be made to provide the programmer with a reasonable model of parallel execution and cache consistency. Several possible models have been proposed [3, 2], e.g., strong ordering, weak ordering, release consistency, etc. Though a full treatment of this topic is beyond the scope of this paper, we will discuss how to add the necessary support to the memory system for a processor to implement a ***fence.*** A fence is a mechanism to "delay the issue of certain accesses until other accesses have been performed," [2] and is the basic operation used to enforce a particular model of execution.

Although each programming model requires slightly different fence semantics, the resulting hardware varies at the processors, not in the memory system. For all models the memory system must be constructed in a way that allows a processor ***to know*** when STORE ***accesses have*** been ***performed*** and LOAD accesses have been ***globally performed*** [3, 2]. By definition, a STORE is performed when a LOAD to the same address by any processor will return the value of the STORE. A LOAD is globally performed when issuing a STORE to the same address cannot affect the value returned by the LOAD, and the STORE that is the source of that value has been performed.

Later when we examine the state transition table for the directory controller we will see that the directory blocks all accesses to a newly-written value until all of the invalidations associated with that write have completed. The new value is therefore not visible to any processor until it is visible to all processors. It is guaranteed, then, that any LOAD to a block by a processor will be globally performed when it completes.

We also need a means for a processor to know when STORE accesses have been performed. In the context of an invalidation-based protocol, a STORE access is performed at the time when all of its related invalidations have completed in their target caches. We must modify the command definitions in our protocol so that a processor can be notified of this condition.

The strategy we adopt is to first have the directory determine when all of the invalidations it has sent have completed in the target caches. To accomplish this ***we*** will now have caches acknowledge each MM→C ***invalidate*** command with an ***invalidate acknowledge (invuck)*** reply. The cache is allowed to send this reply at the point in time when it is clear that the cache will not satisfy another processor reference until after the block is invalidated in the

cache. For instance, if the cache always gives the command input queue priority over the processor, then the *invalidate acknowledge* reply could *be* sent *as* soon *as the invulidute* command is latched into the input queue. But the simplest implementation of the-cache controller will probably not send the *invalidate acknowledge* reply until it begins to process the *invalidate* command.

Note that for Dir_i B and for most simple implementations of Dir_i NB it is possible for a cache to receive an *invalidate* command for a block not currently stored in the cache. The reason is obvious for broadcast schemes: all caches receive broadcast invalidate messages. For protocols without broadcast, this situation is possible because caches typically replace clean blocks by simply invalidating them, without notifying the directory. Thus, the directory information may show that cache contains a clean copy of a block when in fact the cache has replaced it with another block. This is not a problem; if a cache receives such *an invalidate* command, it should simply send a *invalidate acknowledge* reply so that the directory can continue with the assurance that the cache does not have a copy of the block.

Now the commands are in place for the directory to *know* when all of the *invalidate* commands it has sent have been completed. We also need a mechanism for communicating this fact to the cache that sent the *read/exclusive* or *exclusive* command that caused the invalidations. One option is to have the directory delay sending its reply to the *read/exclusive* or *exclusive* command until the invalidations are complete. This scheme has the advantage that no additional messages need be sent; the disadvantage is that the latency seen by the cache is increased since the directory will frequently be able to supply the data before the invalidations have completed Since this penalty can be significant, our design instead sends a separate command to notify the cache that the invalidations are complete. The new MM→C command is called *invalidates done (invsdone)* and no reply from the cache is necessary.

We do not want to indiscriminately send *invalidates done* commands back to *the* cache for every *read/exclusive* or *exclusive* command it issues, because it is not necessary for several common situations. If a cache issues a *read/exclusive* command and the block in question is dirty in another cache, then the reply cannot be sent until the block has been written back anyway. For either a *read/exclusive* or *exclusive* command, the block may not exist in any other caches (e.g., if the block is non-shared *data).* For *these* situations, *we* do not want to send *invalidates done* commands that tie up the network and the target caches needlessly. However, the cache cannot *know a priori* whether or not its command will cause invalidations to occur. To solve this problem we include an extra bit in the encoding of every reply to a C→MM command This bit *indicates* one of two situations: *wait,* in which an *invalidates done* command will be forthcoming from the directory, or *nowuit, in* which no *invalidates done* command will be sent. This is the last modification we need to our commands in order to implement a directory-based cache consistency scheme. The final set of commands in our design is shown in Table 7- 1.

An example of an execution model that can be implemented using these commands is strong ordering [3]. Under this model, a fence operation must occur before issuing a memory reference. The fence delays the reference until all previous LOADs and STOREs by that CPU have been globally performed and performed, respectively [2]. The processor actions needed for implementing this restriction are straightforward Whenever a cache receives a reply indicating the *wait* condition, then no more references from the processor *can be* issued until *an invalidates done* command is received.

The same fence mechanism could instead be used to implement weak ordering [3]. In this case, a fence is not required before each reference, but rather immediately before and after each access to a synchronization variable [2]. Before any references can be issued after the fence, it must *be* known *that* no more *invalidates done* commands are expected from directory controllers. A simple way to ascertain this condition is to use a hardware counter at each cache whose value at any time is the number of expected *invalidates* done commands not yet received. This counter is incremented each time the cache receives a command indicating *the wait* condition, and decremented each time

C→MM commands	Possible Replies
read/ex	retdata/wait retdata/nowait
read/non-ex	retdata/nowait
ex	exack/wait exack/nowait retdata/nowait
writeback	wback
MM→C commands	**Possible Replies**
copyback	cbdata
flush	cbdata
invalidate	invack
invsdone	(no reply)

Table 7-1: Final version of the command messages, revised to include the *wait/nowait* condition needed for the parallel model of execution.

the cache receives ***an invalidates done*** command. Once a fence condition has occurred, further references must be halted until the value of the counter is zero.

8. Directory Controller

Since the purpose of the directory controller is to process C→MM commands, we could not design it until those commands were defined. Now that the command definitions are complete, we can use them as the design specifications for the directory controller. In this section we lay down a basic design by presenting the particular directory organization we focus on, the fields contained in each command, a suitable datapath for the unit, and the state machine that controls the unit.

8.1. Which Directory Organization?

Up to this point in the paper, we have easily avoided deciding what information is stored in each directory entry. This demonstrates that as far as hardware complexity is concerned, the choice of a directory organization really only affects the design of the directory controller itself.

The design we will present is an implementation of Dir_3 NB (3 ⇒ 3 pointers per directory entry, NB ⇒ no broadcast invalidate messages are needed since the number of simultaneous cached copies of any block is limited to 3 [1]). This scheme was chosen primarily because it is a realistic example in the sense that it might be considered for use in an actual machine. We use enough pointers per entry (3) to sufficiently demonstrate the hardware complexity of managing multiple pointers, and the lack of broadcast capability is an accurate reflection of most point-to-point interconnection networks.

For the sake of completeness, let us mention a constraint that would be important if a directory scheme with broadcasts (Dir_i B) were used. The broadcast mechanism must have "all but one" broadcast capability. By "all but

one" we mean the network must support some means of sending **a** broadcast message to all nodes **except for one, where the** one exception is not necessarily the node sending the message. This is necessary in the case **that two** caches issue an ***exclusive*** command on the same block simultaneously. If the directory has to issue a broadcast ***invalidate*** command, then the cache that obtained exclusive access must not invalidate its copy, but the other cache must honor the ***invalidate.*** Since the caches have no way of knowing ***a priori*** if they will obtain exclusive access when they ask for it, they must honor any broadcast ***invalidate*** messages that they see. Therefore, the broadcast mechanism itself must allow the directory to indicate a cache that should not see the message.

In a Dir, NB protocol, each entry contains the information shown in Figure 8-1. There are four fields, namely ***three pointers*** and the ***state.*** The pointers contain the processor numbers of the caches that contain the data corresponding to this directory entry. The number of bits in each pointer is that required to encode a processor number. The state field is comprised of three valid bits and a dirty bit. Each valid bit indicates whether or not the corresponding pointer currently stores a processor number for a cache that has a copy of the block. The dirty bit is set if exactly one cache contains the data, and it is dirty in that cache. For our design we will assume that any one of the three pointers may contain the valid processor number when a block is in the dirty state.

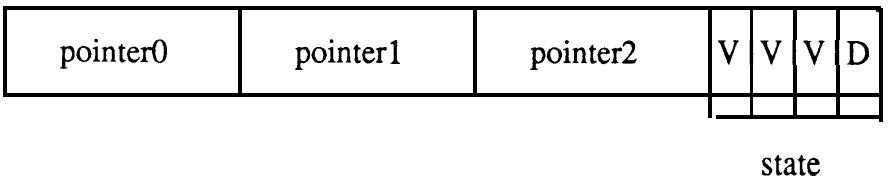

Figure 8-l: A directory entry for the Dir_3 NB scheme.

8.2. Command Fields

Each command must contain several fields of information. Note that some commands may not require all of the fields. The fields are as follows:

- ***command.*** This field indicates which command is being sent. A useful command encoding will probably be one in which separate bits are used to indicate whether it is a C→MM or MM→C message, whether or not the message is **a** reply (the reply bit mentioned earlier), and for replies to C→MM commands, whether or not there is a forthcoming ***invalidates done*** command (the aforementioned ***wait*** or ***nowait*** condition).

***processor number.** The contents of this field depend on whether the command is at the sending or receiving node of the command. At the sending node, this field represents the processor number to where the command will be sent. At the receiving node, the field contains the processor number that sent the command. Of course, on the network the message will contain the node numbers of both the sender and the receiver, but it is expected that the interconnect controllers will add or remove its own processor number to or from the message immediately before it is sent or after it is received. The processor number may be eliminated altogether from arriving reply messages if the directory is single-threaded because the directory will be expecting all replies.

- ***address.*** This field contains the address of the block to which this command applies. In a single-threaded directory implementation, this field need not be included in replies, since a reply always pertains to the same block as the command that prompted the reply.
- ***data.*** This field contains the data words accompanying the command Since a data block is not transferred in every command, this field is not needed in all of the commands.

Each command is therefore made up of several words that contain these fields. The word size is set by the width of the interconnection network paths and the width of the queues. Assuming that the command word size is 32 bits (as well as the addresses and data words in the system), a reasonable command format is shown in Figure 8-2. This defines the ordering of the fields within the command as it passes over the interconnection network and as it sits in the queues. The first word contains the command and processor number fields. The second word contains the address. Subsequent words carry the block of data if the command type warrants it. If each data block is four words long, then commands that carry data will be six words long, and other commands will be two words long.

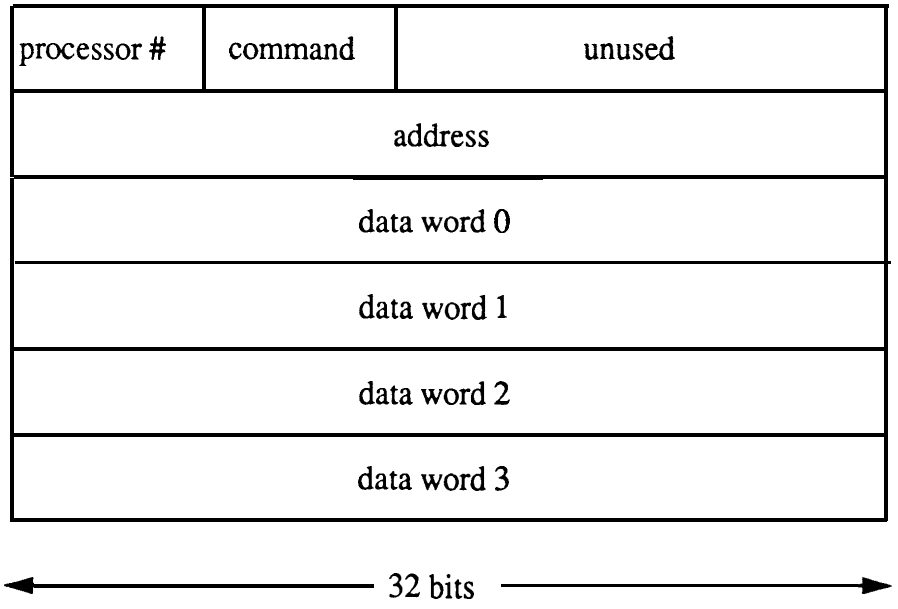

Figure 8-2: The format of each message in the queues.

In some implementations it may be worthwhile to make the command word wide enough to accommodate both the command field and the address field. The advantage of this is that the memory read cycle for the directory state and pointer bits (which must occur regardless of the command) can begin without delay since the address is known. A memory cycle can also begin immediately in main memory proper if the implementation allows the specification of whether it is a read or a write cycle to occur later, and if the cycle can be aborted later if it is found that no memory access is necessary. This may be possible since DRAMs first require the row address to be latched (with the $\overline{\text{RAS}}$ signal) before the data read or write cycle occurs; the ease of implementation will partially depend on the interface with the DRAM controller.

8.3. Datapath

We are now ready to lay down a possible datapath for the directory controller. Looking back at Figure 5-5, the directory simply accepts a stream of commands from one port and produces a stream of commands on another port. We will ***call these the in port*** and out ***port,*** respectively. Our datapath represents logic within the block labeled "DC" in this figure. Though our controller does indicate whether the next command should be removed from a command or reply queue, it does not indicate which of the command or reply queues. The logic used to implement this simple arbitration will depend on the specific nature of the empty/full information provided by the FIFO queues.

The basic datapath is shown in Figure 8-3. Memory is shown abstractly as a "black box" with address and data ports. All data transfer to and from the directory controller occurs on the in and out ports. Keep in mind that all of the fields associated with a command will not appear simultaneously on the in or out port, since the fields sit in different words in the queue. It is the responsibility of the controller to clock fields from the input queue onto the in port in the same cycle that the logic that needs to see those fields is "watching" the in port Similarly, the controller must ensure that the logic drives the out port in such a way that the fields are queued in the correct order at the target queue.

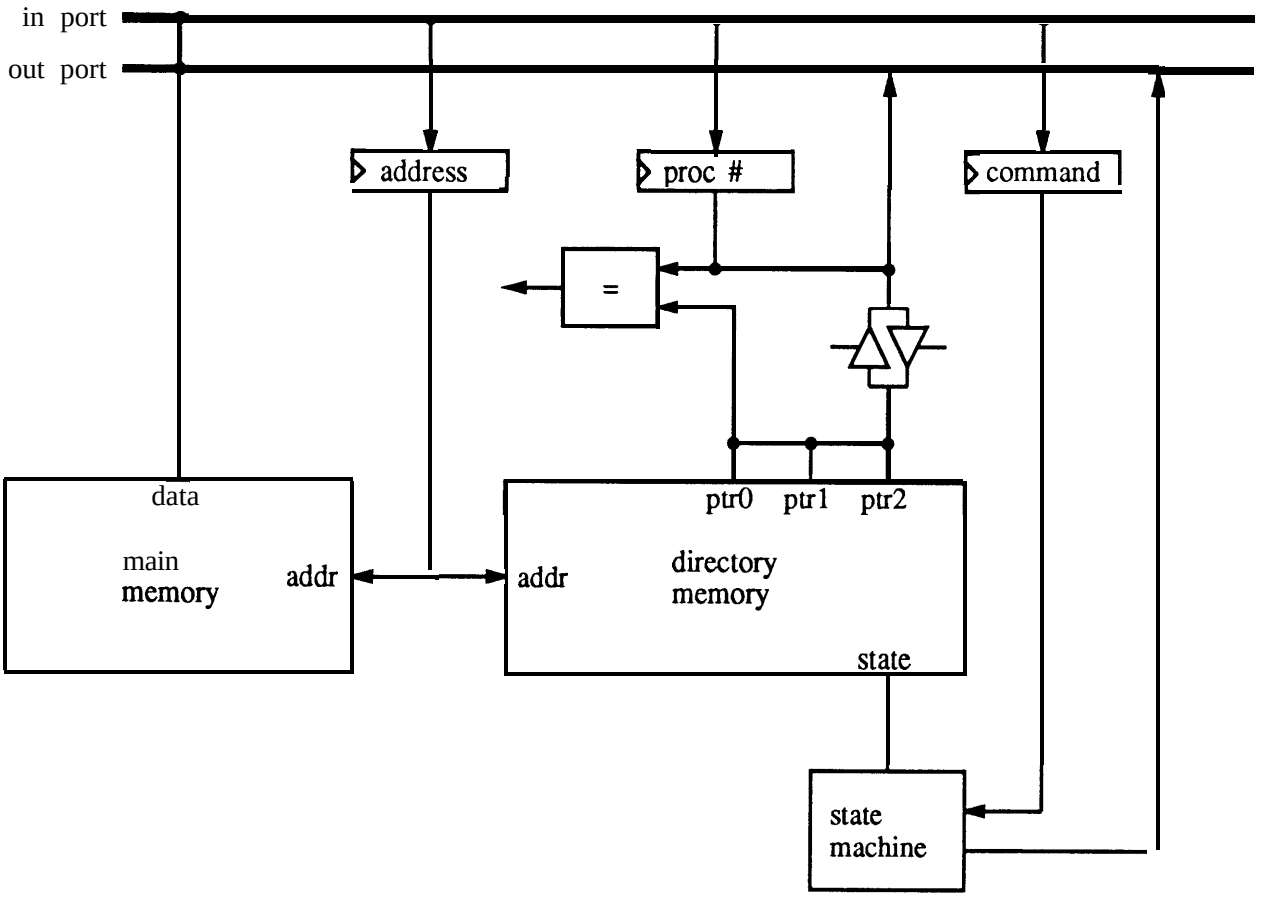

Figure 8-3: The datapath of the directory controller.

Addressing the directory bits and main memory is simple; a single register latches the address from the in port and drives the address ports of the memories. Data blocks are written into memory directly from the in port and read from memory directly onto the out port. The state machine controlling the datapath needs to see both the incoming command type (which is latched in a register) as well as the current state of the addressed block. The state machine also generates the outgoing command type and the next state to be saved in the directory.

A register is included to latch incoming processor numbers. This register should only latch a new processor number on incoming commands, i.e., the register should ignore incoming replies. There are two reasons for this policy. First, as we will see in the state transition table, the processor number from a reply is never needed by the controller. Second, the register must continue saving the processor number from the original C→MM command so that the directory can issue the reply to that command.

The processor number in the incoming command can also be compared against that stored in a pointer. The purpose of this comparator is twofold. First, as discussed before, this comparison is needed to determine if a *writeback* command should be performed or discarded. Second, when the directory sends ***invalidate*** commands in response to a C→MM ***exclusive*** command, the comparator is used to avoid sending ***an invalidate to the*** cache that issued the ***exclusive.***

Outgoing messages can be sent to either the cache that sent the original C→MM command (using the processor number saved in the register) or to the cache indicated by a pointer. The transceiver is needed to isolate the incoming processor number register from the pointers when the comparator is being used.

8.4. State Machine

A rough state transition diagram for a state machine to control the directory datapath is shown in Table 8-l. The diagram is "rough" in the sense that no allowances have been made for timing considerations, such as the number of cycles necessary to access the directory bits. All of the necessary flow of data is indicated, however. Some incoming commands result in multiple outgoing commands; these are denoted with multiple lines in the output side of the table.

8.4.1. Deciphering the Transition Table

The table shows four inputs to the state machine. The first two are the current state and the incoming command type. Another input is the directory state bits for the block specified by the address in the incoming command. Some of the conditions listed in this column are not directly encoded into the state bits (which, recall, consist of three valid bits and a dirty bit) but are easily derived from them with simple combinational logic. These conditions are:

- ***fp. This stands*** for ***free pointer.*** The condition is true if there is at least one unused pointer, i.e., if one of the valid bits is 0.
- ***zero.*** The condition is true if no caches contain the block, i.e., if all of the valid bits are 0.
- one. The condition is true if exactly one cache contains the block and the block is clean in that cache, i.e., if exactly one valid bit is 1 and the dirty bit is 0.
- ***many.*** The condition is true if more than one cache contains the block, i.e., if more than one valid bit is 1.

The final input column ("P# = dirty ptr?") indicates whether or not the processor number in the incoming command matches the single valid directory pointer in the case when the block is dirty.

The first output in the table is the outgoing command type for commands and replies sent by the directory controller. The processor number to which the message should be sent is indicated as well. Due to space constraints, the codes in this column are rather cryptic; we now explain them:

- ***in.*** This indicates the processor number saved in the incoming processor number register.
- *rptr*. This ***stands*** for ***replacement pointer.*** The processor number saved in the pointer that is about to be replaced is to be used. The determination of which of the three pointers this refers to depends on the pointer replacement policy.
- *dptr*. This is an abbreviation for ***dirty pointer. The*** processor number saved in the single pointer that is valid when the block is dirty is to be used. In our design this could be any of the three pointers; the

	inputs:				outputs:						
	state	incoming command	dir state	P# = dirty ptr?	outgoing command	P#	dir state	free ptr	next queue	next state	other actions, comments
1	idle	writeback	dirty	yes	wback	in	zero	*	comm	idle	memory←data
2	idle	writeback	dirty	no	wback	in	*	*	comm	idle	don't write back data!
3	idle	writeback	¬dirty	x	wback	in	*	*	comm	idle	don't write back data!
4	idle	read/nonex	¬dirty ∧ fp	x	retdata/nowait	in	(add)	in	comm	idle	supply data
5	idle	read/nonex	¬dirty ∧ ¬fp	x	retdata/nowait	in					supply data
6					invalidate	rptr	*	in	reply	1	
7	1	invack	x	x			*	*	comm	idle	
8	idle	read/nonex	dirty	x	copyback	dptr	(add)	in	reply	2	
9	2	cbdata	x	x	retdata/nowait	in	*	*	comm	idle	memory←data, supply data
10	idle	read/ex	zero	x	retdata/nowait	in	dirty	in	comm	idle	supply data
11	idle	read/ex	one ∨ many	x	retdata/wait	in					supply data
12					invalidate	ptr0					
13					invalidate	ptr1					only send invs for valid ptrs
14					invalidate	ptr2	dirty	in	reply	3	
15	3	invack	x	x			*	*	reply	4	1st invack
16	4	invack	x	x			*	*	reply	5	2nd invack
17	5	invack	x	x	invsdone	in	*	*	comm	idle	3rd invack
18	idle	read/ex	dirty	x	flush	dptr	dirty	in	reply	6	
19	6	cbdata	x	x	retdata/nowait	in	*	*	comm	idle	supply data
20	idle	ex	one	x	exack/nowait	in	dirty	*	comm	idle	
21	idle	ex	dirty	x	flush	dptr	dirty	in	reply	7	
22	7	cbdata	x	x	retdata/nowait	in	*	*	comm	idle	supply data
23	idle	ex	many	x	exack/wait	in					
24					invalidate	ptr0					only send invs for valid ptrs
25					invalidate	ptr2	dirty	in	reply	8	don't send inv to in P#!
26	8	invack	x	x			*	*	reply	9	1st invack
27	9	invack	x	x	invsdone	in	*	*	comm	idle	2nd invack

Table 8-l: State transition table for the directory controller. An "x" indicates a "don't care" condition, and a "*" means the field should remain unchanged from its previous value.

valid bits must be examined to make the determination.

- *ptr0, ptr1, ptr2.* These refer to the processor number saved in one of the three pointers, pointer0, pointer 1, and pointer2

The next output in the table is the new directory state that should be stored in the directory entry for the block. The only cryptic notation in this column is ***(add),*** which means the valid bit corresponding to the pointer that is becoming valid should be turned on ("added") in the existing state. The next output, "free ptr," shows when the processor number in the incoming command should be saved in a pointer that is free or has been made free by some action taken by the directory controller.

The queue from which the next command should come, either the command queue (abbreviated *comm* in the table) or the reply queue, is indicated. Finally, the next state is listed along with other actions that should be taken.

8.4.2. Transition Table Highlights

There are several points of interest in the transition table, including the subtle cases we presented earlier. In lines 2 and 3 of the table are the cases in which the directory has received a "stale" writeback, i.e., a ***writeback*** command that must ***be*** discarded. ***In*** line 2 the ***writeback*** command comes from a different processor than the processor that currently has the dirty copy of the block as indicated by the directory bits. In line 3 the directory shows the block as clean, ***so the*** received ***writeback*** command must be stale. In both of these cases the command should be ignored and discarded, as discussed earlier.

The other special case we spoke of is when a cache tries to obtain exclusive access to a block with an ***exclusive*** command, but the block is invalidated in the cache before it receives a reply due to another cache performing the same operation. This situation is shown in lines 21-22 of the state transition table. The directory receives an ***exclusive*** command but the directory indicates the block is already dirty, which can only mean another cache obtained exclusive access before this one. The proper action for the directory controller is to flush the data from the cache that has the dirty copy and to send the data back in a ***return data*** reply to the requesting processor.

The cases that may force the directory to send multiple ***invalidate*** commands are when a cache requests exclusive access to a block with either a ***read/exclusive*** or ***exclusive*** command. These cases are shown in lines 11-17 and 23-27 of the transition table. Although it would be simpler for the state machine to send each ***invalidate*** one at a ***time,*** always waiting for the ***invalidate acknowledge*** reply before sending the next ***invalidate, this*** serializes the process, causing it to take much longer than necessary. From a performance standpoint it is obviously preferable to send all of the ***invalidate*** commands with no delay between them and then wait for the ***invalidate acknowledge*** replies to arrive. This is the strategy shown in the transition table. Although ***the*** table shows the ***invalidate*** commands that would ***be*** sent if ***all*** of the ***pointers*** were valid, note that ***invalidate*** commands must be sent using only the pointers that are valid. Also, for the ***exclusive*** command (lines 23-27) ***an invalidate*** command must not be sent to the processor that issued the command, although that processor number will be saved in one of the pointers. The comparator in the datapath is used to detect the pointer corresponding to the processor number in the command; ***the invalidate*** command is suppressed for this pointer. In the example shown in the table, the ***exclusive*** command came from the processor stored in pointer 1; therefore, ***invalidates*** are only sent using pointers 0 and 2. Again, this assumes pointers 0 and 2 are valid.

There are two special cases involving the ***read/exclusive*** and ***exclusive*** commands that allow a slight optimization to be made in the protocol. If a ***read/exclusive*** command is issued for a block that exists in no caches, then there is no need to return the ***data*** with the ***wait*** condition and then later send ***an invalidates done*** command. Since no invalidations need be sent, the data can be returned with the ***nowait*** condition. Similarly, when an ***exclusive*** command is issued for a block that only exists in the cache issuing the command, exclusive access can be granted immediately with the ***nowait*** condition since no invalidations are necessary. These optimizations are shown in lines 10 and 20 of the state transition table. Though they may seem insignificant, these refinements optimize the performance of the memory system for non-shared data, and may be worthwhile for this reason.

Finally, there is one very interesting case that occurs in any directory-based cache consistency scheme that limits the number of cached copies of a block to a given number at a time. This is true for any Dir_i NB scheme (such as our Dir, NB scheme) where i is less than the number of nodes in the system. The situation happens when all of the pointers are valid, i.e., three caches contain copies of a block, and a fourth wants a copy as well to service a read miss. The fourth cache issues a ***read/non-exclusive*** command. This case is unique because it is the only instance in which a read access by a processor forces ***an invalidate*** command to be sent. The purpose of the invalidation, of course, is to limit the number of cached copies to three so that the caches' processor numbers can be stored in the three pointers. But does the reply to the ***read/non-exclusive*** command need to include the ***wait*** condition and be

followed by a ***invalidates done*** command? The answer is no; it is fine to leave a fourth cached copy of a block valid, as long ***as we*** ensure that the block ***is*** invalidated in the cache by ***the time we*** send ***an invalidates done*** command after an ***exclusive*** command is later issued. This is easily accomplished in a single-threaded directory by simply ***waiting*** for ***the invalidate acknowledge*** reply ***to the invalidation*** command before processing the next command from the input queue. In this way we guarantee that the invalidation has completed before any ***exclusive*** command is seen by the directory. Therefore, as shown in lines 5-7 of the state transition table, the correct action to take in this situation ***is*** to return ***the data*** with ***the nowait*** condition, send ***an invalidate*** command and wait for the ***invalidate acknowledge*** reply. No ***invalidates*** done command need be sent.

Unfortunately, this situation in which ***an invalidate*** must be sent to keep the number of cached copies down to three causes a problem similar to the case we studied before when two caches simultaneously issue ***exclusive*** commands. The problem is easily demonstrated by means of an example. Say three caches, A, B, and C, contain clean copies of a block. Now a fourth cache, D, has a read miss on the block and sends a ***read/non-exclusive*** command to the directory. Then cache A wants to write the block, and so it sends an ***exclusive*** command to the directory. The commands are now queued for processing by the directory controller as shown in Figure 8-4. The directory first ***takes*** cache D's ***read/non-exclusive*** command from the queue and decides to invalidate cache A's copy of the block. The directory sends ***an invalidate*** to cache A, cache A invalidates the block, and the directory receives ***the invalidate acknowledge*** reply from cache A. Now the directory controller takes cache A's ***exclusive*** command from the queue. Unfortunately, cache A no longer has a copy of the block.

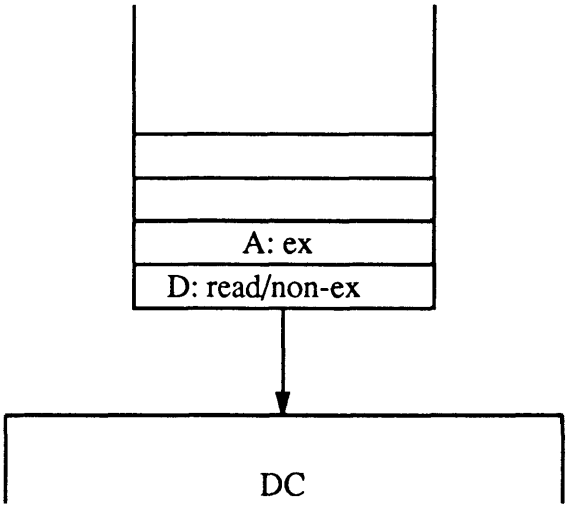

Figure 8-4: The queued commands that will cause the problem.

This situation was easy to detect in the case where two caches issue ***exclusive*** commands at the same time, because in that case the directory shows the block as dirty when the second ***exclusive is*** processed; the directory then knows to return the data rather than simply acknowledge exclusive access. But when cache A's ***exclusive*** command is processed by the directory after its copy has been invalidated, the directory shows the block sitting valid in three caches, just as if there were no problem. The only way to detect the problem at the directory is to compare the processor number of the incoming ***exclusive*** command with each of the stored pointers for that block. If there is no match, then the correct reply from the directory ***is return data*** rather ***than exclusive acknowledge.***

We now present three solutions to this problem; these solutions vary in their trade-off between performance and hardware complexity. Perhaps the simplest solution using the existing datapath is to delay the reply to ***exclusive*** commands until all of ***the invalidate*** commands have been sent. As each pointer is read ***to*** send ***an invalidate***

command, the comparator is used to see if the pointer has the same value as the processor number in the ***exclusive*** command (this comparison is performed anyway to prevent ***an invalidate*** from being sent to the cache that issued the ***exclusive*** command). If no match is found after going through all of the pointers, then the directory replies with a ***return data*** command; otherwise the ***usual exclusive acknowledge*** reply is sent. The obvious drawback of this technique is that the performance of the most frequent case (returning ***an exclusive acknowledge)*** has been compromised to ensure correct operation in the infrequent case.

A second solution is to make the cache "smarter" to recognize the situation when it occurs. In this variant the directory immediately sends the ***exclusive acknowledge*** reply before performing the invalidations. ***As the invalidate*** commands are sent, the comparisons against the processor number in the ***exclusive*** command are performed, as in the first solution. If no matches occur, the directory then sends a second reply to the ***exclusive*** command; this reply is ***return data.*** The cache has to check its tags again whenever it receives ***an exclusive acknowledge*** reply; if the cache no longer contains the block, it ignores the reply and waits for the inevitable ***return data*** reply that will follow. Again, this solution degrades the performance of the frequent case (though not as badly) because of the additional hit detection that must occur in the cache.

We have chosen the solution with the highest performance at a slightly greater hardware cost. The idea here is to add comparators to the datapath so that the processor number of the ***exclusive*** command can be compared against all of the pointers simultaneously. The decision of whether to send ***an exclusive acknowledge*** or ***return data*** reply can therefore be made immediately, and the directory can send the reply before the invalidations. The revised datapath needed to implement this solution is shown in Figure 8-5. A side benefit of this method is that the comparisons with each pointer have already occurred by the time invalidations must be sent so the extra delay time at each pointer before sending the ***invalidate*** command is avoided.

Using the multiple-comparator solution we modify lines 23-27 of the state transition table as shown in Table 8-2. The new input indicates whether or not the processor number in the incoming command matches any of the pointers.

8.5. Implementing Multiple Threads

Up to now we have assumed a single-threaded directory controller for our design. Let us briefly describe the modifications necessary to implement multiple threads. For the most part, the operation of the directory is exactly the same as for a single-threaded implementation, with one important difference. The single-threaded design takes advantage of the fact that the controller sits idle while waiting for a reply by leaving the state needed to complete the processing of the reply in the registers of the datapath and the state machine. With multiple threading, however, those registers must be free to process C→MM commands even while the directory waits for a reply. Our task is to identify the state that must be saved when a MM→C command is sent so we can process its eventual reply, and decide how to save that state.

When a reply from a cache arrives, the directory usually needs to send a reply to the original C→MM message that started the entire transaction. For instance, if the original C→MM message was a request for a block dirty in another cache, and the directory receives the reply from the cache with the ***data,*** then a ***return data*** reply must be sent to the original requesting cache. In order to do this, we need the processor number and address from the original command. This state is easily maintained by simply encapsulating it in the MM→C commands that are sent, with the understanding that the receiving caches will simply echo the state back in their replies. An extra bit must be added to the directory entry as well to indicate that a transaction is pending so that unrelated C→MM messages for that address will not be processed until the pending transaction is complete.

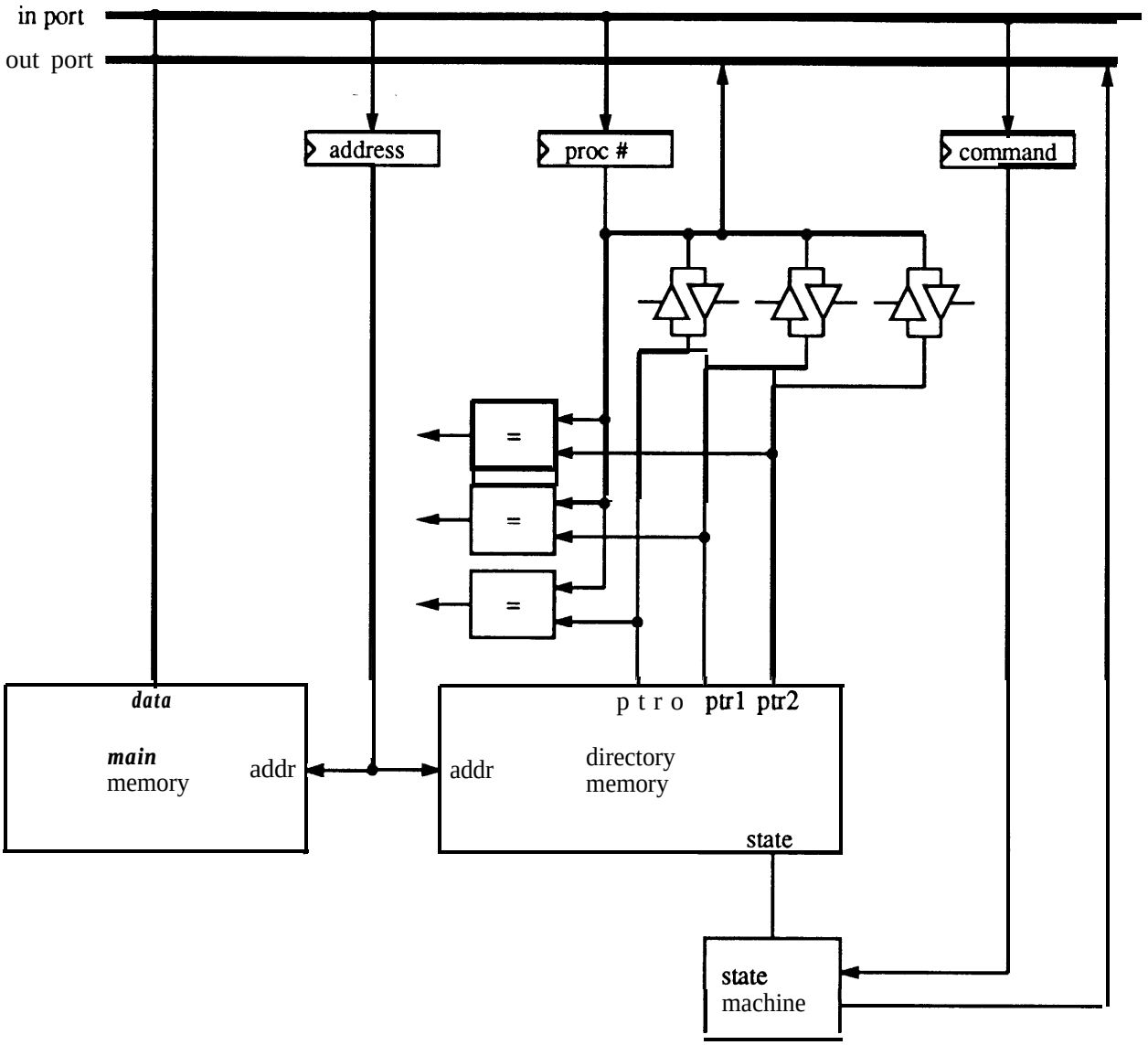

Figure 8-5: The revised datapath includes three comparators so the determination of whether to send *an exclusive acknowledge* or *return data* reply can be made quickly when an *exclusive* command is received.

The only state that cannot be maintained in this manner pertains to the case where the directory must count *invalidate acknowledge* replies *so* that it *can* send *an invalidates* done reply at the appropriate time. This count can be kept only at the *directory since the invalidate acknowledge* replies are collected there. The count could be maintained by adding bits to each directory entry, or by keeping a "counter cache," a memory containing the counter value for each address in a corresponding tag store. In both cases, the counter value would be incremented when an *invalidate* command is sent, and decremented when *an invalidate acknowledge* reply is received. A comparator is also needed so it is known when the counter value reaches zero.

An alternative to using storage dedicated to keeping this count is to simply use the pointer valid bits in the directory entries themselves. The obvious solution of clearing the valid bit of the pointer corresponding to the received *invalidate acknowledge* reply until the valid bits are all zero is slightly flawed. In some cases all cached copies of a block are invalidated (e.g., lines 11-17 in Table 8-1), but in others all but one cached copy is invalidated

	inputs:					outputs:							
	state	incoming command	dir state	P# = dirty ptr?	P# = any ptr?	outgoing command	P#	dir state	free	next ptr	next qstate	other actions, comments
23	idle	ex	many	x	yes	exack/wait	in					
24						invalidate	ptr0					only send invs for valid ptrs
25						invalidate	ptr2	dirty	in	reply	8	don't send inv to in P#!
26	8	invack	x	x	x			*	*	reply	9	1st invack
27	9	invack	x	x	x	invsdone	in	*	*	comm	idle	2nd invack
28	idle	ex	many	x	no	retdata/wait	in					
29						invalidate	ptr0					lines 29-34 same as 12-17
30						invalidate	ptr1					only send invs for valid ptrs
31						invalidate	ptr2	dirty	in	reply	10	
32	10	invack	x	x	x			*	*	reply	11	1st invack
33	11	invack	x	x	x			*	*	reply	12	2nd invack
34	12	invack	x	x	x	invsdone	in	*	*	comm	idle	3rd invack

Table 8-2: Modifications to state transition table to handle problem occurring in Dir_i NB protocols.

(e.g., lines 23-27). A scheme that increments ***as invalidate*** commands are sent and decrements ***as invalidate acknowledge*** replies are received is probably easier to implement. This is still possible using the pointer valid bits, simply by clearing them before sending ***the*** first ***invalidate*** and either setting one bit for each ***invalidate*** or using two of the bits as a counter that can contain the values 0 through 3 (for a Dir, NB scheme).

The extra support we have added for multiple threading can also be used to implement a more efficient solution to the writeback problem we discussed earlier (see Figure 6-2). Because the multiple-threaded directory is built to handle requests while some are already pending, writebacks are easily handled at the directory with no additional hardware. This eliminates the drawbacks of the scheme we mentioned that encodes ***writeback*** commands as replies by setting the ***reply bit*** in the command encoding. All ***writebacks*** will then enter the reply queue at the directory, preventing them from blocking and allowing the directory to reply to a processor that requests the data. The state identifying the requesting processor is contained only in the header of the resulting *copyback* or *flush* command, however, so the receiving cache must send back a reply even if it has already written the data back. A new reply type would indicate that the cache no longer contains the data. When the directory receives this reply with the necessary header ***state, the writeback*** command will have already completed, and the directory can forward the data from memory to the requesting processor.

9. Dir, NB

Only slight modifications are necessary to our Dir, NB design if a $Dir_N NB$[4] consistency scheme is used instead. Whereas the pointers in Dir, NB are fields that contain a processor number, the pointers in Dir_N NB are implemented as a bit vector with N valid bits, one for each node in the system. Two additional pieces of hardware are required in the datapath to handle the bit vector operations.

[4] N is the number of nodes in the system.

First, a unit is needed that accepts the vector of valid bits and produces a stream of processor numbers corresponding to the bit positions in the vector where the bits are set. This unit is used to generate the processor number field for ***the*** stream of ***invalidate*** commands that must be sent by the directory in response to some ***read/exclusive*** and ***exclusive*** commands. Again, we must avoid sending ***an invalidate*** command to the cache that sent the ***exclusive*** command; therefore, as in the Dir_3 NB design, this unit must include a comparator to detect the case when the processor number generated from the bit **vector** is the same as the incoming command processor number register.

Second, hardware must be provided to set **a** single bit in the valid bit vector in the bit position corresponding to the processor number of the current request. The other bits in the bit vector are either cleared or left unmodified, depending on the operation. This functionality is used when a cache loads ***data*** with a ***read/exclusive*** or *read/non-exclusive* command.

Because N cached copies can coexist in Dir_N NB, the situation from the Dir_3 ***NB*** scheme where ***an invalidate*** command must ***be*** sent in response to a ***read/non-exclusive*** command does not occur. We therefore modify lines 4-7 of the state transition table (Table 8-l) as shown in Table 9-l. Note that it no longer makes sense to have the "free pointer" condition as one of the inputs to the state machine.

	inputs:				outputs:							
	state	incoming command	dir state	P# = dirty ptr?	outgoing command	P#	dir state	free ptr	next queue	next state	other actions,	comments
4	idle	read/nonex	¬dirty	x	retdata/nowait	in	(add)	in	comm	idle	supply data	
5												
6												(these lines no longer
7												needed)

Table 9-1: Modifications to state transition table for Dir_N NB scheme.

10. Technology Options

Now that we have completed a basic design for the directory controller, it is possible to evaluate some of the technology options available for implementing the different parts of the system, including the directory memory, the directory controller itself, and the FIFO queues. The goal is to get a feel for the number of parts and amount of board area required for the design.

10.1. Directory Memory

The basic decision to be made for the directory memory is whether to use static or dynamic RAM chips. Static RAM is obviously desirable for speed and simplicity reasons, but it may not be feasible due to its lower density. The amount of directory memory, board area trade-offs, and power considerations will determine whether or not a static RAM directory is practical in a particular system.

Let us look at one sample system configuration, a 256-processor system with 16MB of main memory on each processor node. The caches use 16-byte blocks; therefore, each processor node requires a directory with a million entries. In the Dir_3 NB scheme, 4 bits of state plus three pointers of 8 bits each adds up to 28 bits per directory entry. The directory is therefore made up of 3.5 MB of RAM and will occupy roughly 20% of the total board area used for storage if DRAM is used. For this system configuration, each additional pointer costs about 1.1 MB worth of extra RAM in the directory. For this example, it is clear that the directory cannot be feasibly built using faster but lower-density SRAM.

10.2. Directory Controller

Let us now consider the options for implementing the directory controller. One possibility is to implement the datapath of Figure 8-5 with standard catalog parts and use PALs for the control. As shown in Table 10-1, this datapath requires 12 standard parts (6 registers, 3 comparators, and 3 transceivers), not counting the directory memory. Though we have not explicitly enumerated the state machine's inputs and outputs, the datapath will probably require no more than 3 to 5 PALs to generate the control signals.

unit	bits/unit	parts/unit	# of units	total parts
address register	32	4	1	4
processor # register	≤ 8	1	1	1
command register	< 8	1	1	1
comparator	≤ 8	1	3	3
transceiver	≤ 8	1	3	3
total				12

Table 10-1: Parts needed to implement the datapath for the directory controller, assuming 256 or fewer processors.

Another possibility for constructing the directory controller lies in semi-custom devices such as gate arrays, programmable gate arrays, etc. The directory controller will probably map nicely onto these devices since the unit is relatively self-contained with a manageable number of inputs and outputs (well under 100). In such an implementation, the datapath and control would be combined on the same part. If it is found that the design does not fit on a single part meeting the speed requirements, a bit-slice approach can be used with several parts. Bit-slicing is a clean way of partitioning the design since the only interaction between the bits lies in the comparators, and these are small enough that they would probably not have to be split across chip boundaries.

103. Queues

As earlier mentioned, the FIFO queues in the system are probably best implemented using standard catalog FIFO RAM parts. Since these parts are typically 8 or 9 bits wide, each queue will require 4 parts, assuming a 32-bit command word The seven queues shown in Figure 5-5 will therefore require 28 chips. While this number may seem large, this covers all of the queueing required in the system, not just that needed by the directory controller. Since we count the queue cost here as part of the directory scheme, the design of the cache and interconnect

subsystems is correspondingly simpler since they need not implement any queueing.

11. Conclusion

Starting from a few basic system assumptions, we have worked through a detailed protocol and hardware design of a directory-based cache consistency scheme. We began by focusing on the types of messages in the protocol and the flow of data between the memories and caches. We found that the protocol must be designed in concert with the message delivery mechanisms to remove the potential for deadlock. In particular, the strategy for queueing messages at the input to the directory controller must be carefully planned. By identifying sufficient conditions for preventing deadlock of messages at the cache, at the directory, and in transit, we were able to refine our protocol and hardware to resolve the deadlock problem.

With the basic consistency operations in place, we then added support for a model of parallel execution, such as ***strong or&ring*** or ***weak ordering.*** This provides a reliable programming model even in the face of uncertain latencies for consistency-related operations. To implement such a model, the caches need a way of executing a ***fence,*** and we show that the directory can support this mechanism by counting acknowledgements to invalidations and including a *wait/nowait* bit with each reply to the caches. Using these directory techniques as a foundation, the cache designer can easily implement a ***fence*** operation with the appropriate semantics for the desired execution model.

We then took a more detailed look at the directory hardware, including the datapath required to receive and reply to messages, and the state machine controlling the datapath. Focusing on a system with a limited number of pointers per directory entry, we identified a problem unique to this directory organization that can occur when a memory read causes an invalidation in order to free a pointer. The problem is similar in nature to the more commonly discussed case in which two processors write the same data block at roughly the same time. Several solutions are presented that trade-off performance and hardware complexity differently. We use our final design to approximate the parts count for the major hardware units in the directory.

We have demonstrated that a directory-based protocol is a feasible way of implementing cache consistency across an arbitrary interconnection network. While the resulting protocol is perhaps not simple, it is certainly tractable. In addition, the hardware needed to implement such a protocol is quite reasonable for the scale of machine in which it is likely to be used. Having put concerns about feasibility of implementation to rest, we can now focus on performance issues. The design contained herein should help drive some of these efforts by providing a realistic hardware model for both simulations and analytic models.

12. Acknowledgements

Thanks go to Helen Davis for her good-humored patience and helpfulness in reviewing and discussing this paper. Thanks also to Mark Horowitz for many helpful discussions about the ideas contained herein as they evolved. Dan Lenoski, Mike Johnson, and Kourosh Gharachorloo provided insightful commentary on the first unpublished edition of the paper. Anoop Gupta suggested the technique of counting ***invalidation acknowledge*** messages by turning off the corresponding pointer valid bits in the directory entry. This research has been supported by DARPA/ONR under Contract N00014-87-K-0828.

13. References

[1] Anant Agarwal, Richard Simoni, John Hennessy, and Mark Horowitz.
An Evaluation of Directory Schemes for Cache Coherence.
Proceedings of the 15th International ***Symposium on Computer Architecture ,*** June, 1988.

[2] Kourosh Gharachorloo, Daniel Lenoski, James Laudon, Anoop Gupta, and John Hennessy.
Memory Consistency and Event Ordering in Scalable Shared-Memory Multiprocessors.
Proceedings *of* ***the 17th International Symposium on Computer Architecture ,*** June, 1990.

[3] Christoph Scheurich and Michel Dubois.
Correct Memory Operation of Cache-Based Multiprocessors.
Proceedings *of* ***the 14th International Symposium on Computer Architecture*** , June, 1987.

[4] Wolf-Dietrich Weber and Anoop Gupta.
Analysis of Cache Invalidation Patterns in Multiprocessors.
Third International Conference on Architectural Support *for* ***Programming Languages and Operating Systems (ASPLOS)*** , April, 1989.