AD-A274 026

DTIC
S ELECTE
DEC 23 1993
A

DESIGN OF A
SHARED COHERENT CACHE
FOR A MULTIPLE CHANNEL
ARCHITECTURE

THESIS

John A. Reisner, Captain, USAF

AFIT/GCS/ENG/93D-19

93 12 22 104

93-30991

AFIT/GCS/ENG/93D-19

DESIGN OF A SHARED COHERENT CACHE FOR

A MULTIPLE CHANNEL ARCHITECTURE

THESIS

Presented to the Faculty of the Graduate School of Engineering

of the Air Force Institute of Technology

Air University

In Partial Fulfillment of the

Requirements for the Degree of

Master of Science in Computer Systems

John A. Reisner, B.S.

Captain, USAF

December 1993

To my father, a man of wisdom,

a man who imparts wisdom.

ACKNOWLEDGEMENTS

TABLE OF CONTENTS

LIST OF FIGURES

iv

LIST OF TABLES

AFIT/GCS/ENG/93D-19

ABSTRACT

The Multiple Channel Architecture (MCA) is a recently proposed computer architecture which uses fiber optic communications to overcome many of the problems associated with interconnection networks. There exists a detailed MCA simulator which faithfully simulates an MCA system, however, the original version of the simulator did not cache shared data. In order to improve the performance of the MCA, a cache coherency protocol was developed and implemented in the simulator. The protocol has two features which are significant: (1) a time-division multiplexed (TDM) communication bus is used for coherency traffic, and (2) the shared data is cached in an independent cache. The modified simulator was then used to test the protocol. Two applications and six test configurations were used throughout the testing. Experiment results showed that the protocol consistently improved system performance. Also, a proof-of-concept experiment indicated that performance improvements can be attained by varying cache parameters between the independent shared and private data caches.

DESIGN OF A SHARED COHERENT CACHE FOR A MULTIPLE CHANNEL ARCHITECTURE

CHAPTER 1

INTRODUCTION AND OVERVIEW

1.1. Introduction

A computer's cache is an established necessity for attaining high performance. Large-scale systems presently being designed will need efficient cache coherency protocols in order to realize their potential processing power. One proposed architecture for massively parallel processing is the Multiple Channel Architecture, or MCA. This architecture uses fiber optic technology in an effort to design an ideal interconnection network. The architecture has been tested on a simulator to assess its potential performance. Although much of the MCA has been designed, some elements remain incomplete. For one, the simulator does not cache shared data.

1.2. Cache Memory

Cache memory helps the computer architect diminish the bottleneck between the computer's central processing unit (CPU) and its main memory. CPUs request information from memory at a very high rate--much faster than main memory can provide the data. A cache is a small, fast memory embedded into a system such that memory requests are answered very quickly [Smi82]. Caches are so effective that they have become standard equipment on most computers today [Prz88].

When a computer program accesses a particular address in memory, two things are very likely to happen. First, that address is very likely to be referenced again in the near future. Second, other addresses in the same area

of memory are also very likely to be referenced. Caches exploit these two innate characteristics of software, which are called *temporal locality* and *spatial locality* [Smi82]. Because of these two characteristics, a cache can store a very small subset of main memory, yet still answer a majority of memory requests. A cache hit occurs when the CPU references data already stored in the cache; the slower main memory is used only when the data is not available in the cache (a cache miss). The potential performance gain is dramatic: if a cache is large enough, a 99% hit rate is frequently attained [Smi85, Smi87].

1.3. Cache Coherence

When a CPU writes to a cached variable, the new value must eventually be copied back into main memory. In uniprocessor systems, this is fairly easy to accomplish. Each block of memory in the cache has an associated bit (the "dirty bit"), indicating whether or not the block has been modified. Once the dirty bit has been set, the block must be written back to main memory before it is replaced (overwritten) in the cache [Goo83].

In a multiprocessor system, however, each processor (CPU) has its own cache. This complicates the problem, because more than one processor can cache the same data. Whenever data is shared between caches, steps must be taken to ensure that independent modifications do not occur. As soon as one cache modifies a shared variable, other cached copies of that variable become stale (invalid) until they are updated with the new value. Cached data is said to be *coherent* if the result of any memory read reflects the latest value written to the same memory address [Min92]. This has led to a well-known

problem in parallel computer architecture, the *cache coherency problem* [see Cen78, Pap84, Kat85, Swe86, Che88, Min92]. The solution to this problem is a cache coherency *protocol*, that is, a set of rules which governs the caching process such that all shared data remains coherent.

1.4. Cache Coherence and the MCA

The current MCA uses one of the simplest cache coherency protocols: shared data is not allowed to be cached. Any reference to a shared variable is fetched from memory, and caching is only allowed for data which is *private*, that is, data which is never used by more than one processor. However, the MCA simulator has shown that this method is not efficient enough for implementation. Of the total time spent accessing memory, over 90% of the time was devoted to shared memory accesses, even though these account for less than 10% of the total number of memory accesses [Wai92]. Much of the reason for this imbalance was the high hit rate for private memory accesses (over 99%). However, the architecture needs to be improved by employing a more efficient scheme.

1.5. Thesis Goal

The goal of this thesis was to design a cache coherency protocol which diminishes the excessive time that the MCA spends servicing shared memory requests. Ideally, the designed protocol should be compatible with the architecture, that is, it should exploit the unique aspects of the MCA to its advantage where possible.

1.6. Thesis Overview

Chapter 2 examines existing shared cache coherency strategies, and specific coherency protocols. Chapter 3 provides more background information on the Multiple Channel Architecture (MCA). Chapter 4 proposes a coherency protocol for the MCA, and justifies the design decisions which are made. Chapter 5 explains how this protocol was implemented on the MCA simulator, and how the implementation process refined the design. Chapter 5 also introduces the experiment used to evaluate the protocol. Chapter 6 then presents the results from all experimentation. Finally, Chapter 7 briefly summarizes the findings and provides some suggestions for further research.

1.7. Conclusion

This chapter examined the motivation for caching, and presented a cursory definition of the cache coherency problem. Currently, computer architects are working to design and build massively parallel machines, including the MCA, which could someday connect hundreds of processors and memory nodes [Wai91]. Cache coherency is a major ingredient for successful implementation of systems this large. Some solution to the coherency problem must be developed for the MCA, in order for the MCA to be a legitimate architecture proposal. The purpose of this thesis is to propose, simulate, and evaluate a cache coherency protocol especially suited for the MCA.

CHAPTER 2

CACHE COHERENCY PROTOCOLS

2.1. Cache Coherency

Memory is said to be *coherent* if the value returned on a read is identical to the value stored during the most recent write to the same address [Min92]. A coherency *protocol* is a set of rules used to ensure data remains coherent in a system. Keeping data coherent in a multicache system is now a well-known problem; several solutions have been proposed and implemented. Generally, these solutions fall into three categories: *snooping protocols, directory protocols,* and *software protocols.* For these protocols to work, the system must have some way of knowing which caches contain a given word (or block) of data. In a snooping protocol, caches monitor the transactions between main memory and other caches by "snooping" a common bus. Directory protocols use a set of pointers (the "directory") to track which caches contain each block of memory. In a software protocol, the code is analyzed to determine when shared memory may be safely cached.

Coherency protocols can be simple or complex. A more complex protocol may yield better performance, but at the cost of requiring more circuitry or memory.

2.2. Attributes of Cached Data

Several factors determine what happens after a processor makes a memory request. Two of the more obvious factors are the type of request (read or write) and whether or not the data resides in a local cache (cache hit

2.1

or cache miss). These two factors define four basic types of transactions: read hit, read miss, write hit, and write miss.

Each data block involved with a transaction has a number of attributes; a combination of these attributes can determine the state of the data.

The first attribute is data *privacy*. If data can only be accessed by one processor, it is considered *private*. If it can be accessed by more than one processor, then it is *global*. The coherency problem is relevant only to global data; private data requires no coherency protocol. Sometimes the word *shared* is used in place of *global*. However, the term *shared* is also used in conjunction with exclusiveness (the fourth attribute, which is explained below); therefore, in this chapter, the word *global* will be used to describe non-private data.

The second attribute is data *validity*. If memory or a cache contains the correct, up-to-date value of the data, then its block is *valid*. If the block has been updated elsewhere, then the outdated copy of the data is *invalid*, or *stale*.

The third attribute is data *cleanliness*. If the value of cached data reflects the value contained in main memory, then the data is *clean*. If the data in the cache has been updated, and main memory has invalid data, then the cached data is considered *dirty*. Cleanliness differs from validity in that validity is concerned about the most up-to-date value of a variable, while cleanliness is concerned only with comparing cached data to the value stored in main memory at a particular time. Dirty data must eventually be written back to main memory. The terms *modified* and *unmodified* are sometimes used in place of clean and dirty.

The fourth attribute is *exclusiveness*. If global data resides in one and only one cache, then it is considered *exclusive*. If the data possibly resides in

2.2

more than one cache, then it is considered *non-exclusive,* or *shared.* It may be possible for a shared data block to be the only copy across the caches. For example, this would occur when a shared block is initially found in two caches, and then one of the caches replaces that block. The remaining shared data block remains shared until it is proven to be the only copy found in the caches of a system.

The fifth attribute is *ownership.* The *owner* of a data block is responsible for ensuring the data remains coherent. Protocols which employ ownership allow exactly one owner per data block. All data is initially owned by main memory, but ownership can be transferred to a local cache. Some protocols only allow the owner to modify a block, so a cache must explicitly acquire ownership before writing to a block. Data residing in a particular cache can be classified as owned or unowned by that cache.

Coherency protocols define what actions take place during a memory transaction, depending upon the states of the data blocks. Whenever a potentially shared block is modified, a *notification* takes place. A notification can either be an update (where all cached copies of the variable are provided with the latest changes), or an *invalidation* (where other cached copies of the block are invalidated).

Because four Boolean attributes have been defined for global data, up to 16 distinct states are possible. However, many of these can be ignored by a protocol. For example, ownership applies only to valid data (the owner of the data must have a valid copy). Some protocols have been developed which use as few as three different states [Arc86].

2.3. Sweazy and Smith's MOESI Model

Sweazy and Smith [Swe86] have defined a class of coherency protocols which account for five possible states. The first state is INVALID (I); the four remaining states apply to valid data. These are: SHARED-CLEAN (S), EXCLUSIVE-CLEAN (E), SHARED-MODIFIED-OWNED (O), and EXCLUSIVE-MODIFIED (M). This model is entitled the MOESI model, derived from the letters used to abbreviate the five states. The attribute of cleanliness is ignored—all modified data is owned and all unmodified is unowned (owned by main memory). Blending cleanliness and ownership into one property does simplify the model by reducing the total number of states. However, it also adds inherent restrictions. Only the owner can modify a cache block, and, similarly, ownership must be transferred before a non-owner can modify a block. Because some coherency protocols have this restriction, this model may need special adaptation to fit a particular protocol.

The MOESI model is not a particular protocol per se, but a model which represents a configuration of attributes which can be used to design a protocol. The following three sections describe actual protocols which have been implemented or proposed.

2.4. Examples of Existing Snooping Protocols

2.4.1. The Synapse Protocol

The Synapse protocol was one of the first coherency protocols successfully implemented [KSR92]; it is used in the Synapse N + 1 multiprocessor.

There are three states in Synapse: INVALID, VALID, and DIRTY [Arc86]. VALID data is clean, unowned, and potentially shared. DIRTY data is

owned. Therefore, Synapse does not account for the attribute of exclusiveness. The three states used by the Synapse represent the minimum number of states required to make a snooping-based protocol work [Kat85].

Synapse requires each tag in main memory to have a "dirty bit," which indicates whether or not the data block is DIRTY in some cache. The following paragraphs describe what happens for each possible transaction [Fra84]:

READ MISS, NO CACHE HAS DIRTY COPY: The block is simply fetched from main memory. All other caches with valid data retain their VALID state for this block.

READ MISS, A CACHE HAS DIRTY COPY: The original read request is aborted, since the dirty bit indicates that the copy in main memory is not valid. The owning cache then writes back the block to main memory, and changes its local state to INVALID. The original requester then makes another request, which this time takes place. The requesting cache receives the data in the VALID state.

WRITE MISS, NO CACHE HAS A DIRTY COPY: The requesting cache obtains the block from main memory, while all other caches with this block change their block status to INVALID. The requesting cache receives the data in the DIRTY state.

WRITE MISS, A CACHE HAS A DIRTY COPY: The cache with the dirty copy sends the data to the requesting cache. Main memory does not get updated. The cache which previously held the data now changes its block status to INVALID. The requesting cache obtains the block with a status of DIRTY.

READ HIT, IN A DIRTY/VALID CACHE: The data is read, and processing continue:

WRITE HIT, IN A DIRTY CACHE: The block is updated, and processing continues.

WRITE HIT, IN A VALID CACHE: This is handled like a write miss, where no cache has a dirty copy. The block is actually fetched from main memory again, and the states are updated as described in the write miss with no dirty copy.

2.5

2.4.2. Comments about the Synapse Protocol

Synapse maintains coherency, but there are some tradeoffs worth noting. Some questions arise when examining the efficiency of the protocol. The first is, "Why read the block from main memory on a write hit, when the status is valid?" Synapse only transfers ownership during a write miss. Under this rule, a write hit can occur only in a cache where the block is already owned (DIRTY). The protocol was designed such that there is only one way to transfer ownership, at the cost of requiring an additional read on a WRITE HIT.

Another question is: "Why change the state to INVALID after a read miss, when the cache had a dirty copy?" The cache which had the dirty block changes its state to INVALID after the update to main memory [Fra84, Arc86]. But the data in the cache is still valid! A change to the INVALID state is only necessary when the requesting cache is performing a write, not a read. However, by always changing the dirty block to INVALID, there is no need for the relinquishing owner to discern between reads and writes, thus simplifying the protocol.

2.4.3. The Berkeley Protocol

The Berkeley designers wanted to minimize the number of bus transactions required to access shared data [Kat85]. To do this, they added an additional state to the Synapse protocol, the OWNED-EXCLUSIVE state. The Berkeley states are: INVALID, UNOWNED, OWNED-EXCLUSIVE, and OWNED-NONEXCLUSIVE. Thus, Berkeley accounts for the attributes of validity, exclusiveness, and ownership, but it does not account for cleanliness. The owner alone has the right to update a block. Therefore, it is presumed that an owned block is a modified block.

The Berkeley cache is capable of two types of reads and two types of writes [Kat85]. The two types of reads are Read-Shared and Read-for-Ownership. Because ownership is required for data writes, the Read-for-Ownership provides a mechanism for acquiring ownership from main memory prior to a data write. Read-for-Ownership is also used when accessing private data, thereby eliminating the need to explicitly acquire ownership at write time. A Read-Shared is used to cache global data without acquiring ownership.

The two types of writes are Write-for-Invalidation and Write-without-Invalidation. When all of the caches (which are snooping) detect a Write-for-Invalidation, any cache which contains the modified block updates the status of that block to INVALID. The write to main memory does not take place on Write-for-Invalidation; main memory is only updated during a Write-without-Invalidation. Write-without-Invalidation is used for flushing owned blocks back to main memory upon replacement and ownership transfers. As the title implies, all caches with valid copies retain their state during a Write-without-Invalidation.

Before a non-owning cache can modify a block, the cache initiates an ownership transfer. Ownership transfers take place via the following steps:

(1) The previous owner flushes the block back to main memory, using a Write-without-Invalidation.

(2) The cache requesting ownership then obtains the block with a Read-for-Ownership.

The following paragraphs explain how the Berkeley protocol handles the various transactions, depending upon the state of the data [Kat85]:

SHARED-READ MISS, NO OTHER CACHE HAS OWNED COPY: The block is fetched from main memory and put in the cache with the state UNOWNED.

SHARED-READ MISS, ANOTHER CACHE HAS OWNED COPY: The request is serviced by the owning cache, which gives the block to the requesting cache with the state UNOWNED. If the owner's state was OWNED-NONEXCLUSIVE, no other actions are required. If the owner's state was OWNED-EXCLUSIVE, then the state in the owning cache is changed to OWNED-NONEXCLUSIVE.

READ-FOR-OWNERSHIP MISS, NO CACHE HAS A COPY (This action normally occurs when accessing data known to be private): The block is fetched from main memory, and put in the cache with the state OWNED-EXCLUSIVE.

READ-FOR-OWNERSHIP MISS, ANOTHER CACHE HAS UNOWNED COPY (This action normally occurs as part of a write miss, which causes an ownership transfer): Main memory gives the block to the requesting cache, with the state of OWNED-EXCLUSIVE. All caches with an unowned copy update their status to INVALID.

WRITE MISS, ANOTHER CACHE HAS OWNED COPY: Ownership is transferred as already described (block is flushed back to memory by previous owner, followed by a Read-for-Ownership by requesting cache). The end result: all previously cached copies of the data become INVALID, and the new owner obtains the data in the OWNED-EXCLUSIVE state.

WRITE MISS, NO OTHER CACHE HAS OWNED COPY: The block is fetched using a Read-for-Ownership. When fetched, it will have the state of OWNED-EXCLUSIVE, and the write may transpire with no additional bookkeeping.

READ HIT, IN A VALID CACHE: The data is read, and processing continues.

WRITE HIT, IN AN OWNING CACHE: If the state is OWNED-EXCLUSIVE, the block is modified locally, and processing continues (no bus transaction required). If the state is OWNED-NONEXCLUSIVE, the block is updated with a Write-for-Invalidation. The state is changed to OWNED-EXCLUSIVE in the owning cache, and all other caches which held the block update their state to INVALID.

WRITE HIT, IN A CACHE WITH UNOWNED COPY: Because the writing cache must obtain ownership, this situation mirrors a write miss. If the block is owned by another cache, then ownership is transferred. If no cache owns the block, then the first step of the ownership transfer can be skipped, and ownership is obtained using a Read-for-Ownership.

2.8

2.4.4. Comments about the Berkeley Protocol

Recall that the Synapse protocol requires three bus transactions when a modified block is read by another cache. These three transactions are:

(1) A cache requests the block from main memory. This request is aborted when the dirty bit is found to be set.

(2) The cache with the dirty data writes the block back to main memory.

(3) The first cache re-requests the data. This time the request is granted.

The Berkeley designers specifically wanted to reduce the number of transactions required in this situation [Kat85]. They accomplished this by allowing the owner of a block to service a read request, while retaining the dirty copy of the data. Synapse cannot allow this, because it lacks a state which indicates that data is both dirty and shared. The OWNED-NONEXCLUSIVE state provides this information.

An added advantage of this state is that it allows the protocol to differentiate between shared and exclusive data. Therefore, exclusive data can be modified without having to notify caches of the update. This keeps the bus free from meaningless broadcasts.

The protocol may unnecessarily burden the bus during ownership transfers, however. If one cache can provide dirty data to another, without writing back to main memory, then why can't that cache also grant ownership without writing back to main memory? Instead of first writing the block to memory, and requiring the new owner to perform a Read-for-Ownership, the initial Write-without-Invalidation could be changed. It could instead invalidate other copies of the data, while at the same time granting ownership to the new owner. This would save one bus transaction per ownership transfer.

2.4.5 *The Dragon and the Firefly*

The two protocols previously discussed allow only one writer per block, namely, the owner. There are, however, coherency schemes which are not limited by this restriction. Two such schemes are the Firefly protocol (used in DEC's Firefly multiprocessor) and the Dragon protocol (used in the Xerox PARC Dragon). These schemes are very interesting in that they do not have an INVALID state--all cached data remains valid. Whenever a write is performed, the writing cache sends the modified data to the other caches, rather than an invalidation signal.

These two protocols require a "SharedLine" on the bus [Arc86]. Whenever a block is accessed, caches which contain the block assert the SharedLine. Thus, exclusiveness is detected at block access time.

Archibald and Baer [Arc86] report three states for the Firefly: EXCLUSIVE, SHARED, and DIRTY. Therefore, this protocol only accounts for two attributes: exclusiveness and cleanliness. By definition, DIRTY data is exclusive, and SHARED data is clean. When a write to a SHARED block occurs, the writing cache transmits the modified word over the bus. Main memory and other caches with the data then update their copies of the word. This configuration works as follows [Arc86]:

READ MISS, NO CACHE CONTAINS COPY: When no other cache asserts the SharedLine, the block is fetched from main memory. It is loaded in the EXCLUSIVE state.

READ MISS, ONE OTHER CACHE CONTAINS EXCLUSIVE COPY: The SharedLine is asserted, and the block is furnished by the cache with the data. Both caches now have the block in the SHARED state.

READ MISS, ONE OTHER CACHE CONTAINS DIRTY COPY: The dirty data is transmitted on the bus, where it is received by both the requesting cache and main memory (main memory gets updated). The result is that the two caches now contain the SHARED block.

2.10

READ MISS, OTHER CACHES HAVE THE SHARED BLOCK: The block is provided to the requesting cache by a cache which already has the data. All caches retain the SHARED state.

WRITE MISS, NO CACHE CONTAINS COPY: The block is loaded from main memory in the DIRTY state.

WRITE MISS, OTHER CACHE/S CONTAIN/S EXCLUSIVE/SHARED COPY: The block is loaded in state SHARED. The writing cache will broadcast the write, so that both main memory and all other caches with the block will remain valid.

WRITE MISS, ONE OTHER CACHE CONTAINS DIRTY COPY: The cache with DIRTY data writes the block onto the bus, updating main memory and providing the requested data to the other cache. The two caches now contain the block in the SHARED state; when the second cache modifies the data (performs its write), the modified word will be put on the bus, so all other caches (and main memory) will remain valid.

READ HIT: The data is read from the cache, and processing continues.

WRITE HIT, IN A SHARED CACHE: The write is transmitted over the bus, so all copies of the data remain valid.

WRITE HIT, IN AN EXCLUSIVE OR DIRTY CACHE: If the state was previously EXCLUSIVE, it is updated to DIRTY. The write takes place and processing continues with the block in the DIRTY state.

The Dragon has four states: EXCLUSIVE-CLEAN, SHARED-CLEAN, EXCLUSIVE-DIRTY, and SHARED-DIRTY. Therefore, the states can be represented with and CLEAN bit and a SHARED bit. The advantage of the Dragon's extra state (SHARED-DIRTY) is that writes to main memory can be delayed--when a cache modifies a data block, the modifications are sent only to the caches which share the data. Only one cache can have a block in the SHARED-DIRTY state, and that cache is responsible for updating main memory before the dirty block is replaced.

Dragon accounts for exclusiveness and cleanliness. The attribute of ownership is accounted for, also, since only one cache can have a block in a

2.11

DIRTY state. That cache is responsible for (1) informing the caches which share the block of all modifications, (2) eventually updating main memory (before the block is overwritten), and (3) furnishing the data to caches which request it after a miss.

The Dragon transfers ownership much more easily than the Berkeley protocol. If a cache has a write hit, it simply transmits the change over the bus. The cache which broadcasts the write also updates its block state to DIRTY (SHARED-DIRTY if the SharedLine is raised). All other caches perform the update, and retain the state of SHARED-CLEAN. The cache which made the most recent update will always have the block in the DIRTY state (that cache becomes the new owner), and all other caches will use the state SHARED-CLEAN. This is, in fact, a misnomer, since the data is actually dirty (it does not reflect the stale value in main memory). However, the scheme works to delay a write to memory for as long as possible.

2.4.6. Comments on the Firefly and the Dragon

Of the six snooping protocols surveyed by Archibald and Baer [Arc86], the Dragon had the best performance, and the Firefly was a close second. It appears that DEC modified the Firefly coherency scheme some time after the [Arc86] evaluation, and has since adopted a scheme more like the Dragon, based on an article published by Thatcher [Tha88]. The protocol described in [Tha88] has the same four states as Dragon, and it behaves similarly.

In uniprocessor systems, the primary aim of the cache is to decrease memory access times for the processor. An effective snooping coherency protocol must not only try to keep processor access times low, but also minimize bus traffic. Caching is a very important part of bus saturation prevention.

Both Firefly and Dragon do this by transmitting updates to other caches, rather than invalidation signals. By keeping data valid while it is shared, there are fewer times a block has to be re-fetched from memory. Dragon further attempts to lighten the processor load on main memory by deferring copy-backs for as long as possible [Tha88].

2.5. Overview of Directory Protocols

The coherency protocols previously discussed are all members of the snooping family, where caches snoop a common bus to maintain block states. Unfortunately, snooping protocols can saturate the common bus if the number of processors gets too high (20-30 processors) [Aga88, Fra84]. The snooping protocols already presented are therefore not suitable for large parallel systems, because the protocols are *not scalable*. A system or subsystem is considered scalable if, as the number of processors increases, the performance improvement of the system is linear or near-linear [Len92]. All the protocols described were designed to run on systems with one common bus, and such an architecture is not considered scalable. In order for a snooping protocol to be scalable, some improvement would need to be made, such as a hierarchical bus scheme.

Directory protocols track data states (validity, exclusiveness, cleanliness, and ownership) by maintaining a set of pointers, rather than by monitoring broadcasts over a shared bus. The set of pointers which tracks this information is called the directory. Directory protocols were proposed as early as 1978 [Cen78], but they have recently gained popularity since they are considered to be better suited for larger parallel systems [Ste90]. [Cha90] describes three types of directories: limited, fully-mapped, and chained.

2.13

A limited directory is set up so that only a fixed number of processors can share the same block. The directory size is N bits per block, where 2N caches can simultaneously share the same block. The main disadvantage to limited directory schemes is that a cache may be forced to give up a block if the number of processors requesting the block exceeds the imposed limit. Theoretically, limited directories are not scalable if the number of processors simultaneously accessing global data increases as the number of processors increases.

A fully-mapped directory is large enough to track its data blocks, even if a block resided in every processor's cache. The size of a fully-mapped directory is N bits per block, where is N is the number of nodes interacting with the memory module (normally, N = number of processors, but this could vary with a hierarchical cache scheme). Fully-mapped directories can be quite large--size is one of their chief disadvantages [OKr90]. Another potential problem is the changing of directory hardware every time the number of processors increases.

A chained directory is distributed in the caches themselves, rather than in a central location. Chained directories are much like a linked list. The directory in main memory points to one cache with the data. That cache has a pointer which points to another, and so on. Chained directories do not limit the number of sharing processors, and the size of the directory is independent of the number of processors. The main disadvantages are the "ripple delay" which occurs when traversing the list, and the overhead of list maintenance (such as when a processor in the middle of the list has a block replacement).

A fourth type of directory protocol is an overflow directory [Ste90]. Overflow directories are very similar to limited directories, except that when

2.14

the directory overflows, the overflow directory uses a secondary method to maintain coherency. This secondary notification scheme can vary between architectures. For example, an overflow bit could be used to indicate that all caches need to be notified of any change to a particular block. Another scheme might use a pointer (the start of a chained directory) once all other directory entries are used.

2.6. The Stanford DASH Directory Protocol

The Stanford DASH architecture is a set of clusters. Each cluster contains a small number of RISC processors and is allotted a dedicated portion of main memory. Clusters communicate over an interconnection network. The DASH coherency directory works among four levels of a memory hierarchy, described below [Len92]:

Level 0 (Processor level). This level includes the processor and its local cache.

Level 1 (Local cluster level). This level includes all the processor caches within the cluster.

Level 2 (Home cluster level). This is where all main memory and the cache directory are located. The cache directory has one bit for each cluster, rather than for each processor, reducing its overall size.

Level 3 (Remote cluster level). This level includes all processor caches from remote clusters.

The DASH coherency directory is located in main memory. It has three states for data blocks: UNCACHED, SHARED, and DIRTY. SHARED data is always clean, and DIRTY data is always exclusive (the DASH protocol combines cleanliness and exclusiveness, much like the original Firefly protocol combined these two attributes). Additionally, the local cache maintains state bits which indicate validity and cleanliness. Thus, validity is

2.15

detected at the local cache. Exclusiveness and cleanliness can be detected either at the directory or at the local cache.

The DASH protocol interfaces with the cluster architecture as follows: when a reference is made, the local cache is checked (Level 0 check). If there is a hit, the request is serviced at that level. If there is a miss, or if further action needs to be taken, the transaction is forwarded to the next level. This process continues through the levels until the transaction is complete (the process will always complete at or before Level 3).

During a read and write, the following activities occur at each level [Len92]:

READ FROM MEMORY:

LEVEL 0: If the local cache has the data, processing continues. Otherwise, a read miss is propagated to Level 1.

LEVEL 1:   If a cache within the local cluster contains the data, this cache provides the block to the requesting cache. No state change is required in the directory, since the data block remains within the cluster. If the block is not in the cluster, the request propagates to Level 2.

LEVEL 2:   If the block is clean (i.e., UNCACHED or SHARED), then it is immediately sent to the requester, and the directory is updated to show that this cluster now has the data. If the directory indicates the block is DIRTY, then the request is forwarded to Level 3.

LEVEL 3:   The cache with the dirty data sends the block to both the requesting cache and main memory. The two caches will then have SHARED data, and the directory is updated to reflect this.

WRITE TO MEMORY:

LEVEL 0:   If DIRTY data is in the cache, the write proceeds without delay. Otherwise, the request is propagated to the next level.

LEVEL 1: If this cluster has the block in the dirty state, the dirty block is transferred to the requesting cache. The directory and main memory remain isolated from the transaction. Otherwise, the write is forwarded to Level 2.

LEVEL 2: If the directory indicates the block is SHARED or UNCACHED, the write request is granted. The state of the writing cluster is set to DIRTY; any other clusters with the data receive invalidation signals. If the block is DIRTY in another cluster, the write request is forwarded to that cluster.

LEVEL 3: The dirty data is sent from the remote cluster to the requesting cluster. An update is sent to the directory, so that it knows the new possessing cluster. The sending cluster also invalidates its old copy of the data.

BLOCK REPLACEMENT:

A dirty block is only written back to main memory upon replacement. If the block belongs to the memory associated with the local cluster, there is no need to transmit over the interconnection network.

2.7. Comments on the DASH Protocol

The coherency protocol benefits from the cluster architecture. Two objectives in parallel systems are (1) to minimize latency (i.e., keep the miss wait time low), and (2) increase the memory bandwidth (i.e., keep main memory as free as possible).   By allowing the local cluster to handle transfers within the cluster, the DASH accomplishes both objectives. Additionally, on a miss to a dirty block, the remote cluster responds directly to the requester, rather than writing through main memory. This also reduces latency and increases the bandwidth.

Another benefit of clusters is the reduced size of the directory. DASH has a fully mapped directory without having to use one bit per processor. Processors can be added to existing clusters without changing the directory size.

2.8. Overview of Software Protocols

Software coherency protocols rely upon code analysis to determine whether or not particular data blocks are cacheable during a given program epoch. An epoch is defined as either a parallel loop in a program, or a serial section of code between such loops [Min92]. The simplest software protocol is one which simply will not allow a shared variable to be cached, thereby ensuring coherency. Another option is to disallow shared variable caching only during program epochs when writes occur to that variable or block. This requires a more complex analysis, but it can lead to improved performance. In general, performance may be improved as th. .ompile-time analysis grows more complex.

Without run-time checks, software protocols must take a conservative approach when determining what is and is not cacheable (an access which may be stale at run-time is treated as miss, whether or not the data has actually been modified). This degrades performance, since conditional writes reduce the hit ratio, even when the writes are not performed during execution. However, the hit ratio is only one important element of a coherency scheme. Another important facet is the amount of bus traffic incurred while maintaining coherency. This becomes critical in massively parallel systems. Because software protocols do not rely on run-time communication, they are potentially more scalable than both snooping protocol and directory protocols.

Most software protocols only deal with the attribute of validity. Cache entries are either valid or invalid. Because of this, many software schemes use write-through, which keeps the number of states at two.

One early software protocol was designed so that the *entire cache* was invalidated whenever a possible stale access was encountered [Che90]. Invalidating an entire cache seems so drastic that such a protocol would be of no practical use. However, Cheong and Veidenbaum claim that such a scheme can achieve "good performance," in spite of its obvious fault [Che88]. [Che90] shows some test results using this protocol, and the performance was reasonable for four out of the seven simulation applications they tested ("reasonable" in this context is loosely defined as achieving a hit rate of at least 50% of the ideal hit rate).

More recently proposed software protocols attempt to overcome previous shortcomings in two areas. First, they use *selective invalidation*, that is, they invalidate only a subset of the blocks after a given write. Second, they attempt to combine some run-time results with the compile-time analysis to avoid unnecessary invalidations. Two such schemes are those by Cheong and Veidenbaum [Che88], and Min and Baer [Min92].

2.9. Cheong and Veidenbaum's Fast Selective Invalidation Protocol

During the compile-time analysis, each memory reference is marked as either a *memory-read* or a *cache-read*. Cache-reads are used whenever the cached variable is guaranteed to be valid, based on the software analysis. Cache-reads are always serviced by the cache. Memory-reads are serviced by memory whenever the run-time analysis indicates that the cached variable has become stale.

Each entry in the cache has a *change bit*, which determines whether or not a memory-read to that entry is treated as a hit or as a miss. This bit is set *for every entry in the cache* each time an INVALIDATE CACHE instruction is

executed. INVALIDATE CACHE instructions are inserted by the compiler; they appear in conjunction with code segments which possibly perform memory writes, thereby causing extra cached copies to become invalid.

If every entry in the cache gets its change bit set, then where are the savings over indiscriminate invalidation? There are two scenarios where performance gains are realized. The first is when a cache-read is used (instead of a memory-read). In this case, the data is fetched from the cache, regardless of the change bit. The second scenario occurs when fetching an item from memory, since the change bit is reset. Thus, additional references which occur before the next INVALIDATE CACHE are serviced by the cache, even though the reference is marked as a memory read.

2.10. Comments about Cheong and Veidenbaum's Protocol

Although an improvement over previously proposed software protocols, Cheong and Veidenbaum's fast selective invalidation protocol still does not *eliminate* unnecessary invalidations. It has no provision for determining whether or not conditional writes actually occur, and thus still takes the conservative approach for each instance of a conditional write. For example, consider the following segment:

```
    if cond1 then
            Update (Array_X)
    end if;
    if cond2 then
            Update (Array_Y)
    end if;
    Update (Array_Z)
            :
            :
    /* invalidate cache */
L1:    Read_from (Array_X)
```

The invalidate cache is inserted because some modifications occurred in the previous epoch. Since Array_X may have been written to during that epoch, then the Read_from statement at L1 must be flagged as a memory read, not a cache read. Therefore, the scheme will always fetch from main memory at L1, regardless of whether or not cond1 evaluated to true. If cond1 was false, then the data in the cache would still be valid.

Despite this shortcoming, this protocol is one of the better software protocols, and was chosen as one of the best schemes for comparison purposes when Min and Baer proposed their clock-and-timestamp protocol [Min92].

2.11. Min and Baer's Clock and Timestamp Protocol

In this scheme, each shared data structure (array or scalar) has a counter, or clock associated with it. Also, each cache block has an associated *timestamp*, its length in bits equals that of the clock. All clocks are initially set to 0, and each variable's clock is incremented between two program epochs where the associated data item *may* have been modified. Whenever a word is loaded into the cache, the timestamp is updated based upon the value of that variable's clock at load time. A cached item is considered valid if the value of the timestamp is greater than or equal to the current value of the clock. The end result is that a cached variable is considered valid until the start of a program epoch where the cached value may become stale.

2.12. Comments about the Clock and Timestamp Protocol

Min and Baer claim that this protocol is able to deliver a better hit ratio than other software protocols. However, it does require more hardware.

Namely, a timestamp is required for each cache block, thereby increasing the size of the cache. Min and Baer point out that this additional storage requirement only grows in relation to cache size; whereas, in a fully mapped directory, the storage requirement grows in relation to main memory—a much worse situation. But the timestamp is not free, either. Min and Baer suggest a 16-bit timestamp associated with each cache block.

The one place where Min and Baer's scheme has an advantage over other schemes is that it keeps track of which processor modified the data so that the writing processor will have a cache hit the next time it references that item. Whenever a processor writes to a shared variable, and then reads it again before the next epoch, the timestamp protocol will have an additional hit. The more frequently this occurs in a program, the bigger the performance gain. The question is: "Does this event happen often enough to justify the more complex hardware requirements?"

Another good idea introduced in their scheme is selective loading of the cache. If there is no way that a data item can be used before it becomes invalid, it is not loaded in the cache. For example, suppose one processor (processor A) reads a data item X. Suppose also that before processor A references X again, processor B modifies X. There is no need to load X into processor A's cache, because X cannot be used, and it might replace data which would have otherwise caused a cache hit. Any coherence protocol—snooping, directory, or software—could benefit by such a compile-time analysis.

2.13. Summary

This chapter examined five cached data attributes. Each attribute can be applied when defining the states used in a coherency protocol.

The first attribute was data privacy. An example of a privacy protocol is the software scheme which prohibits the caching of global data.

The second attribute was data validity. The DASH and Berkeley protocols are examples of invalidating protocols. These protocols send an invalidation signal to caches with copies of dirty data. The Firefly is not invalidating; it keeps all data valid by transmitting the updated value, rather than in invalidation signal.

The third attribute was data cleanliness. A cleaning protocol would be one which always employed write-through, so that main memory always remained valid. None of the snooping protocols discussed in this paper are cleaning protocols; all of them allow dirty data in the caches. Many software protocols keep cached data clean so that any cache with invalid data can simply get an updated copy from main memory.

The fourth attribute was exclusiveness. An excluding protocol would allow only one copy of a global variable to be cached at any given time. Such a scheme would be inefficient, and none of the schemes discussed in this paper fit this description.

The fifth attribute was ownership. The Berkeley protocol is a classic example of an ownership protocol; only the owner of a block is allowed to modify that block. In contrast, the Firefly does not account for ownership.

The five attributes are normally combined to simplify the protocol. .For example, exclusiveness and cleanliness are often integrated, as in the DASH protocol. Combining can vary greatly between protocols: the Dragon protocol keeps these same two attributes mutually exclusive.

Some combinations will blur the distinctions between the attributes. When dirty data must be kept exclusive, is the exclusive holder also the

2.23

owner? In this chapter, ownership has only been applied to schemes which allow more than one dirty copy in the caches.

This chapter also described the three families of protocols and detailed at least one example of each. Some benefits and disadvantages associated with the three approaches were discussed.

One area of extreme concern is scalability. Because of bus saturation, snooping protocols are not considered scalable. Snooping protocols are also limited because they must monitor a common bus. The Wisconsin Multicube has actually implemented a snooping protocol despite having more than one bus [Goo88], but snooping all the network traffic would be very difficult for an architecture with many channels, such as the MCA. Directory protocols are better, but fully-mapped directories are generally size-prohibitive for large systems. Chained directories have no size problem, but there is a communication overhead associated with maintaining and traversing the chain. A good software protocol can be scalable, since it analyzes the software independently of the target hardware platform. However, since it is difficult to detect whether or not a possible data write will actually occur, unnecessary invalidations are a problem.

A way to effectively enforce coherency may be to use a memory hierarchy, and use different types of protocols at each level. The Stanford DASH does this in their cluster architecture, using a directory at main memory, and a snooping protocol at the cluster level [Len90]. Additionally, Min and Baer point out that their software protocol may be more effective in a multilevel cache system, with a directory protocol at the next level [Min92].

2.24

CHAPTER 3

THE MULTIPLE CHANNEL ARCHITECTURE (MCA)

3.1. The Multiple Channel Architecture

As mentioned in Chapter 1, the chief goal of this thesis is to design a cache coherency protocol for the Multiple Channel Architecture, or MCA. In order to fully understand the protocol design decisions, the MCA must be explained. This section briefly discusses the MCA: why it was designed and how it works. For more detailed information, see [Wai91], [Wai92].

The MCA is a proposed computer architecture. It uses an alternative communication system which overcomes many of the obstacles in massively parallel systems. Namely, it uses fiber optic communication paths and tunable lasers in lieu of a conventional interconnection network. Because different frequencies (channels) can be simultaneously broadcast on the same fiber optic path, many different messages can be sent over the same "wire" at one time.

The ideal interconnection network allows any node to communicate with any other node in one step. However, a fully connected topology is difficult to realize because the complexity of the network increases dramatically as nodes are added to the system (see Figure 3.1). To overcome this problem, various strategies have been proposed and implemented. One strategy is to configure the nodes in a way that allows communication in a relatively small number of steps. As an example, the hypercube allows one node to communicate with any other in no more than $\log_2 n$ steps, where n is the number of nodes in the network. Another strategy is to implement some type of switching network. The system is configured so that the nodes can always communicate in one step, by traversing a network of switches. A chief

3.1

disadvantage is that many times, two messages can not be broadcast simultaneously, because they would need to be routed through the same switch (see Figure 3.2).

Because the MCA uses fiber optic communication paths in conjunction with tunable lasers, communication between any two nodes can be done in one step.

Each node has at least one laser receiver/transmitter (called an *r/t*, for short). For the duration of a particular task, each r/t is tuned to receive on a *fixed channel*. The transmitters are tunable, and can be tuned to a given frequency (channel) in a few nanoseconds [Wai92]. If node A wants to communicate with node B, then node A first tunes its transmitter to the channel on which node B receives, and then sends its message (see Figure 3.3). [Wai91] reported that each channel is likely to reach a 1 Gb/sec bandwidth. Systems with a 3 Gb/sec bandwidth and as many as 100 distinct channels have already been successfully tested [Gla93, Kar93]. The MCA simulator assumes a 2 Gb/sec transmission rate.

As currently planned, the MCA has three types of nodes: memory, I/O, and processor. Memory nodes and processor nodes have two r/t's. One is a conventional r/t, and the other is a time-division multiplexed r/t which can be used for rapid barrier communication. A wide variety of configurations can be realized by various combinations of channel sharing. Research to date shows that using one channel per node results in gross underutilization. Therefore, processor nodes and memory nodes are likely to share the fixed receive channels.
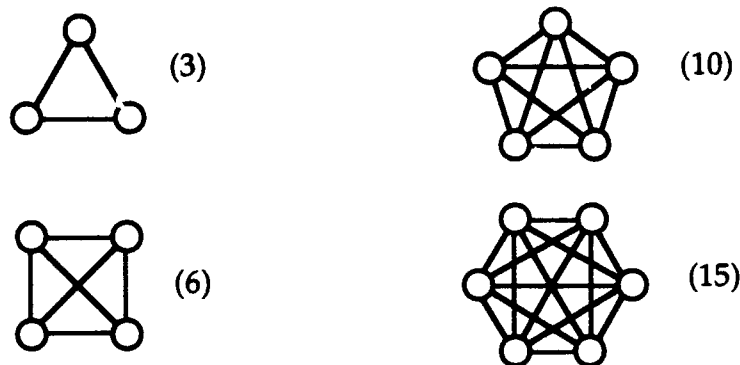
Figure 3.1: Fully Connected Topologies. Fully connected topologies for systems with three, four, five, and six nodes. The number of connections required for each is listed to the right. For a 16-node system, 120 connections would be required.

SOURCE
NODE

DESTINATION
NODE

Possible switch states are:

straight
through

cross

upper
broadcast

lower
broadcast
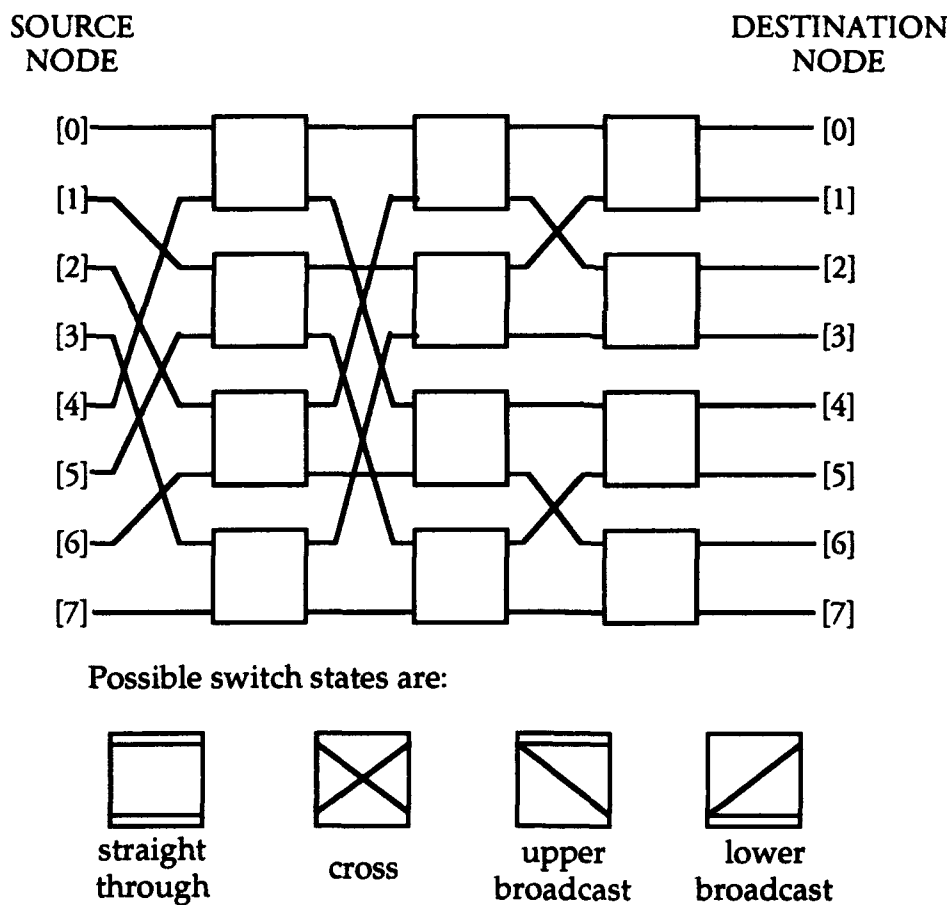
Figure 3.2: Generalized Cube Network. A generalized cube network with eight nodes. Node 0 can transmit to Node 1 while Node 6 transmits to Node 7. However, Node 4 cannot transmit to Node 2 while Node 0 transmits to Node 1, because there is a conflict at the top left-hand switch.

3.3
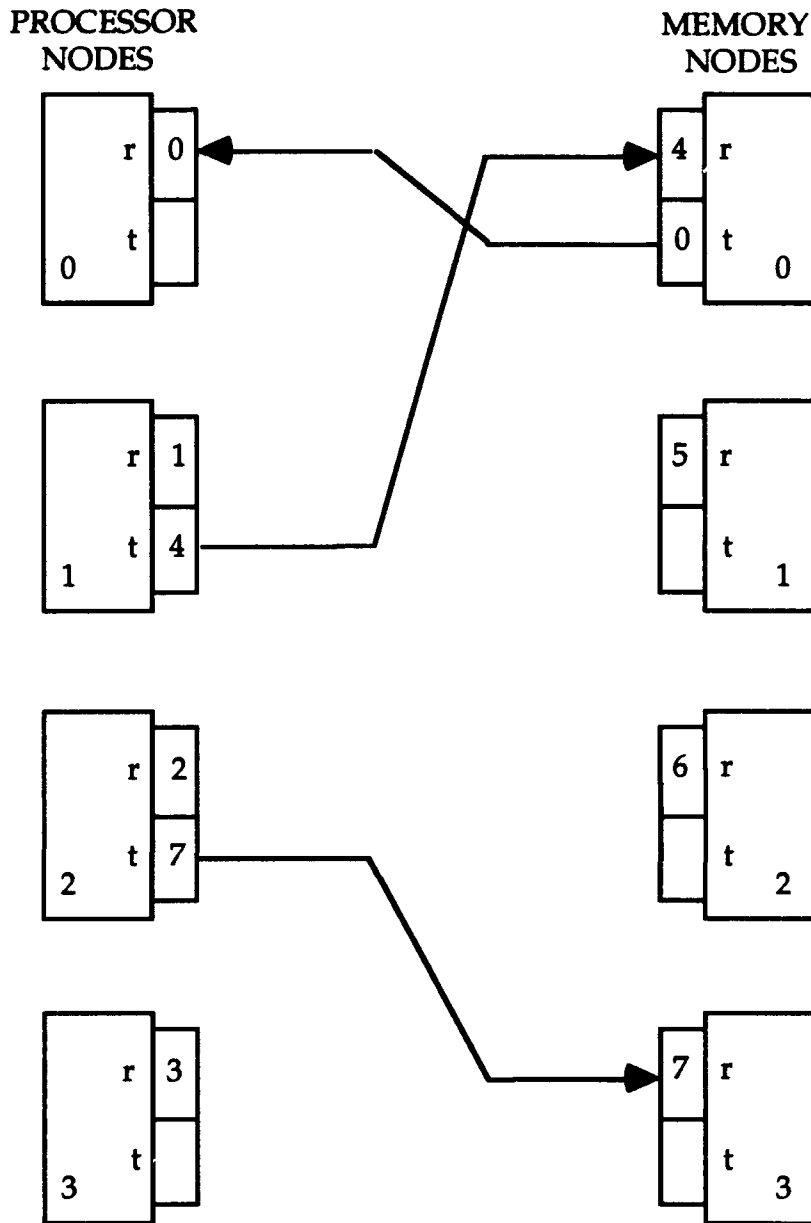
Figure 3.3: MCA Communication Example. An MCA configuration with
four processors and four memory nodes. The processors are tuned to
channels 0 through 3; the memory nodes are tuned to channels 4 through 7.
To communicate with a particular node, the sending unit tunes its
transmitter to that node's receiving channel, then transmits. In this example,
Processor 1 is transmitting to Memory 0, Processor 2 is transmitting to
Memory 3, and Memory 0 is transmitting to Processor 0. These transmissions
can all occur simultaneously.

3.2. The MCA Simulator

In order to evaluate the performance of the proposed architecture, a simulator has been built [Wai92]. This is a program written in the PCP computer language, a parallel version of C. The program attempts to faithfully simulate the MCA, under a variety of configurations, running a suite of parallel programs. The test suite currently includes four programs. MATRIXMULT multiplies two square matrices whose size is determined as a parameter at run-time. FILTER performs 64x64 median filtering. GAUSS performs gaussian elimination on a square matrix whose dimension is given as a parameter at run-time. WATER performs a simulation of water molecule movement (the number of molecules simulated is a run-time parameter). A modified version of this simulator is the foundation for the experiment of this thesis. These modifications and the experiment are discussed in Chapter 5.

At run-time, MCA configuration information is passed to the simulator. This information includes:

The number of processor nodes (CPUs).
The number of shared memory nodes.
The number of private memory nodes.
The number of channels to be used by the processors.
The number of channels to be used by shared memory nodes.
The number of channels to be used by private memory nodes.
The overlapping of channels (one channel can be shared by multiple
    processors, shared memory nodes, and/or private memory
    nodes).
The number of tasks to be run during the simulation (each task has its
    own configuration of processors, memory nodes, and channels,
    although channels can be shared between tasks).
The size of the private cache (defaults to 64K).
Other cache parameters (defaults to a 4-way set associative private cache
    with a line size of 32 bytes).

The simulator is event driven, not trace driven. The memory controllers, busses (channels), cache controllers, I/O ports, and processors all have associated finite state machines which ensure that the simulation remains true.

As an example, suppose CPU 0 needs to fetch an instruction. First, the cache controller for processor 0 must be free. If it is, it checks to see if the address of the instruction is contained in a valid block in the cache. If the line is not cached, then the CPU must form a request packet. This packet contains the information required by the IEEE 802.3 (Ethernet) Standard, including a preamble, start, destination, source, length, type, data field, and cyclic redundancy check (CRC). A built-in delay is associated with the packet formation; the delay time varies according to the length of the packet. After the delay, the status of the bus is checked. If it is busy, the packet must wait until the bus is free. Once the bus is free, an attempt is made to transmit. If there is a collision with another packet, the collision is resolved using a CSMA/CD protocol with a binary exponential truncation limit of 10. Once the packet is on the bus, the bus remains busy for a set time, based on the propagation delay and modulation rate for a five-meter cable length. When the packet arrives, it is put into the memory controller's receive buffer. It will be serviced from there once the controller is free. There is an associated delay with this servicing. After the reply packet has been formed, it must be put onto the bus in the same manner as it was at the processor node. The transmit time is dependent upon the length of the packet. After the processor receives the packet, the line can be put in the cache only when the cache controller is free. The CPU sends an acknowledgement packet back to memory for every packet it receives; these packets also have delay times and must go through the same bus arbitration. After the line is loaded in the

cache, the CPU can fetch the instruction. If the instruction requires data which is not is the cache, a cache line must be retrieved from memory again. CPUs stall whenever (a) an instruction miss occurs, and (b) whenever a second miss to the same address occurs. Condition (b) prevents out-of-order execution of the same variables. Out-of-order execution does occur when the same variables are not involved. The stall conditions are intentionally very conservative.

The simulator was designed to maximize faithfulness to an actual MCA, in terms of delays and network traffic. If cache ejections of private data are required, an LRU (least recently used) scheme is used, with a write-back packet required whenever the ejected line is dirty. This write-back packet will prompt an acknowledgement packet from memory after it is received.

The simulator also keeps track of many performance statistics. Totals are kept of the number packets requested and sent by each node, the number of collisions on each bus, the bus utilization, the CPU utilization, the cache hit rates, the number of cache ejections, and literally dozens of other items. Additionally, periodic histograms of most of these statistics are available so that performance trends of this data can be identified and analyzed. For example, a low cache hit rate during the first 50,000 clocks would be expected, since the cache is "cold" during program initialization. Similarly, more collisions would be expected during this period, particularly if memory nodes are sharing channels, since many more requests are being made to the memory nodes.

3.3. Summary

This chapter discussed the MCA architecture and the MCA simulator. Knowledge of the architecture is important in order to understand the design

decisions described in the next chapter. A modified version of the MCA simulator incorporates the protocol described in Chapter 4, and this version was used to test the protocol and measure its performance. This experiment is described in Chapter 5.

CHAPTER 4

PROPOSED COHERENCY PROTOCOL FOR THE MCA

4.1. Selecting a Coherency Protocol for the MCA

The goal of the MCA cache coherency protocol is to balance necessary tradeoffs such that the overall design has several desirable characteristics. The ideal protocol is scalable, and it has a high hit ratio for various multiprocessing applications. Notifications (invalidations or updates) are completed as quickly as possible. The ideal protocol requires minimal communication between nodes and minimal hardware overhead. A "smart" protocol would not cache variables which will not be referenced again and never fetch from memory when a valid value is already stored in the cache.

4.1.1. Applying Snooping Protocol Principles to the MCA

Snooping protocols, as described previously, are not suitable for ensuring MCA coherence. There are too many channels in the MCA architecture for snooping to work. However, the snooping principle can be used as part of the overall coherency scheme. For example, only one transmission is required to notify several processors tuned to the same channel. Whenever multiple processors are tuned to the same channel, they are, in effect, all snooping a common bus.

4.1.2. Applying Directory Protocol Principles to the MCA

Directory protocols are potentially more scalable [Gup92], and are a strong candidate for the MCA coherency strategy. The MCA may be particularly suited for a directory protocol, since a directory could be maintained based on channel, rather than processor. In this scenario, a fully-

mapped directory would only require one bit per channel, rather than one bit per processor. Processors tuned to the same channel would snoop together.

4.1.3. Applying Software Protocol Principles to the MCA

Software analysis can be used to make any protocol more efficient. The following paragraphs mention some generic ways that compile-time analysis can improve any hardware protocol's performance, including a protocol for the MCA.

First, compile-time analysis could prevent caching of a variable that is not read by a processor before the next modification. In this situation, the compiler could designate this reference as a *memory-only reference*. This prevents possible replacement of a valid block in the cache with a block that will be invalidated before a cache hit ever occurs. In this situation, the memory node could send one word rather than an entire block, thereby reducing communication traffic. The one area where this needs to be studied further is when other entries in the cache block would be referenced before the invalidation occurs (for block sizes greater than 1). In such circumstances, the memory-only references may actually degrade performance.

A second means to improve a protocol is exploit *write-runs* [Egg88]. A write-run is the number of times that the same processor writes to a variable before another processor accesses that variable (see Table 4.1). In such cases, there is no need to commence the notification process until after the last write of the run, potentially reducing message traffic. Eggers and Katz [Egg88] report that the most write-runs are of length 1 or 2; however, some write runs were so long that the average length was over 5 for two of the four applications they traced. If a compiler marked a write-run of length 20, it would save 19 unnecessary notification broadcasts--a significant savings. Two

```
WRITE_TO (VARIABLE_X) by PROCESSOR A ⌉
                  ⋮                      Processor  A  alone
                                         accesses  Variable_X
WRITE_TO (VARIABLE_X) by PROCESSOR A     during  this  code
                  ⋮                      segment.

WRITE_TO (VARIABLE_X) by PROCESSOR A
                  ⋮                   ⌋

READ FROM (VARIABLE_X) by PROCESSOR B     This access by
                                          Processor B ends the
                                          write-run.
```
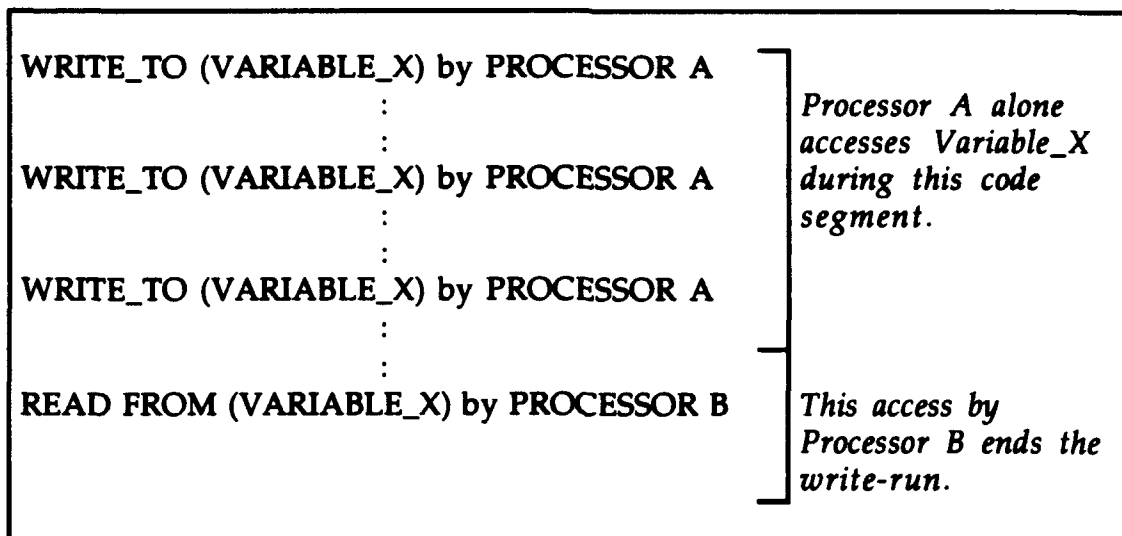
Table 4.1: An Example Write-run of Length 3.

of the four applications analyzed in [Egg88] had over 6% of their write-runs of length 20 or more.

Third, some shared variables are modified one time only, at the beginning of the program. After this one write occurs, the data is no different from private data, and will never be invalid from that point onward. Depending upon how a coherency scheme is developed, this information might also make a protocol more efficient.

4.2. Two Caches per Processor—Separating Private and Shared Data

Using two caches per processor on a parallel machine has many potential advantages over a unicache system which caches both global and private data. These benefits are detailed in the following sections.

4.2.1. Eliminating Coherency Hardware Associated with Private Data

Unfortunately, in a unicache mode, the overhead required by a particular protocol is applied to all cached data, even though this overhead

4.3

may be meaningless for private data. For example, consider a snooping protocol with four states for cached data. Each cache block requires two bits to track the state. If 70% of the data in the cache is private, than these two bits are only relevant for 30% of the cache (since the private data remains valid). The chip area used for the two bits, as well as the logic used to verify validity at reference time, is, in a sense, wasted.

If a processor had two caches, one dedicated to private data and the other dedicated to global data, then this waste could be eliminated. The private cache would contain no state bits or validity verification logic—only the global cache would.

What percentage of the cache is used for shared data? Gupta and Weber analyzed five parallel programs, and found a large variation in the percentage of references to global data (vs. private data) [Gup92]. Eggers and Katz [Egg88] used trace statistics, and also found a wide range of values (see Table 4.2). However, these values only reflect the percentage of *references*, so they do not indicate volume. If shared variables are referenced more frequently on a per variable basis, then the percentage of shared data in the cache would actually be less.

Assuming that the shared and global data are referenced evenly on a per variable basis, we can use the mean value of 43% in Table 4.2. Assuming only 50% of cache references are for data (the remaining 50% for code, see [Smi82]), then we can generalize: (.43) x (.50) = 22% of all cache entries in a unicache system contain shared data. Of course, due to the wide range of results in Table 4.2 and the broad assumptions made, we must understand that 22% is at best a typical figure, with a fairly large fluctuation possible. Nevertheless, it seems safe to say that, at any one time, less than half of a cache is likely to

4.4

Application	Shared References	Private References	Source
Maxflow	57%	43%	[Gup92]
MP3D	76%	24%	[Gup92]
Water	17%	83%	[Gup92]
PTHOR	48%	52%	[Gup92]
LocusRoute	53%	47%	[Gup92]
PLOVER	39%	61%	[Egg88]
PSPICE	24%	76%	[Egg88]
PUPPY	31%	69%	[Egg88]
TOPOPT	42%	58%	[Egg88]
(Mean)	43%	57%	

Table 4.2: Percentages of References to Shared vs Private Data. Percentages are of references to data, and do not include code references.

contain shared data, even though the coherency hardware overhead is present for every entry.

4.2.2. Customizing Cache Parameters for Two Different Types of Data

How does system performance vary as cache parameters are changed? This issue has been studied several times [Smi82, Smi85, Arc86, Smi87, Prz88, Hil89, Prz90]. Parameters of interest include line (block) size, overall cache size, set associativity, and replacement policies. In parallel systems, the shared data and private data are cached the same, so optimal parameter values may in fact be a compromise. The biggest issue is whether or not shared data and private data have the same characteristics. For example, is spatial locality equally true for private and shared data? If spatial locality applies less to shared data, then it would make sense to use a smaller block size for shared data. A block size such as 32 is very effective for private data, but it may be too big for shared data.

By separating shared data into a separate cache, its characteristics can be studied, ultimately leading to set of optimum parameters. If the best shared data parameters exactly match the optimum parameters for private data, then varying these parameters between separate caches would probably yield little performance benefit. But if ideal parameters varied, cache customization could improve system performance. There is no reason that the shared cache could not have a different block size or replacement policy, and there is no reason that a shared cache could not be designed smaller or bigger than the private cache.

4.2.3. Copy-back vs. Write-through

In systems which have no global data, a copy-back scheme will perform better than write-through, since multiple writes to a cached variable do not need to be forwarded to memory until the block is replaced. The decision to copy-back or write-through in a multicache system, however, is more complicated. Whenever a global variable is modified, the writing cache needs to notify the system, unless the affected block is exclusive and main memory is allowed to remain invalid. But even if the cache can postpone notification, it will need to get involved the next time the modified block is requested by another cache. Because of this complexity, a write-through scheme has some appealing features in a parallel environment, such as the ability for main memory to always answer a cache miss. By separating the private and global caches, a copy-back scheme can be used for private data, while a write-through scheme can be used in the global cache.

4.2.4. Reducing Side Effects of Shared Data Updates

When shared data is modified, actions are taken to ensure coherency. Some protocols use block invalidation, such that each invalidation affects an entire block. Such protocols could benefit from a shared cache with a smaller block size, since smaller blocks would theoretically (1) decrease data sharing, thus reducing invalidations, and (2) decrease the amount of data invalidated with each invalidation. Additionally, invalidations and updates add undesirable processing overhead to the cache. For example, a cache may have to delay a code fetch while a coherency update is being processed. By separating private data, this added burden does not affect the caching and fetching of instructions and private data.

4.3. A Proposed Directory-Based Shared Cache

The proposed shared cache coherency protocol for the MCA has two caches per processor. One cache contains only global data; private data has its own, separate cache. Every memory reference is either global or private. So a memory reference causes a lookup in one of the two caches, depending upon which type of data it is. The private caches require no coherency protocol: line size, overall size, replacement policy, and organization can be set up any way which optimizes performance.

The shared cache maintains coherency by using an overflow directory. Chapter 2 mentioned that fully-mapped directories are size-prohibitive, and that limited directories may perform badly because any time the number of processor clusters needing a block exceeds the hard limit, a conflict occurs. Either the requesting processor is denied permission to cache the variable, or

else another processor cluster must "give up" the block. Neither option is very appealing.

In an *overflow directory*, an overflow bit is associated with each block, in addition to the directory assigned to each block. The directory has a length of n, where n is the number of channels which can be stored in one directory entry. If processors tuned to n or fewer channels have cached a given block, then the directory alone maintains enough information to enforce coherency. When processor (n + 1) requests access to the block, the overflow bit is set. This indicates that some secondary notification scheme must be used.

As mentioned in Chapter 2, the secondary notification scheme can vary between architectures. In lieu of an overflow bit, a pointer could point to the beginning of a chained directory.

4.4. The MCA Overflow Directory Coherency Protocol

The MCA overflow directory is channel-based, rather than processor-based. These directories (one directory per global cache block) reside in the memory nodes. When a processor modifies a shared block, the processor alerts the owning memory node of the change. If the overflow bit is not set, the memory node sequentially transmits the new information across each channel listed in the directory. Thus, updates are used when the directory is not overflowing.

If the overflow bit is set, then a global notification takes place. During a global notification, every cache is notified that a particular block has been modified. On the MCA, global notifications will be performed via an *Extended Invalidation Broadcast*, or EIB. During an EIB, *every* processor is provided with the address of the modified block, and those processors which

have a cached a copy of the block must invalidate it. This can be done rather simply by using the barrier channel mentioned in Section 3.1 as an "EIB channel." Recall that an extra, time-division multiplexed r/t exists at each node. At every processor, these r/t's are all tuned to the *same channel*. When a memory node sends a message over the EIB channel, all of the processors will receive it. In essence, the overflow bit forces the MCA to revert to a snooping scheme for ensuring coherency. However, the primary disadvantage of snooping schemes (bus saturation) is overcome because the EIB channel is used exclusively for invalidation messages. These messages, therefore, do not interfere whatsoever with regular network traffic.

It is possible that the EIB channel itself will saturate. To prevent this, independent tasks running on the MCA could be assigned different EIB channels. Because of the large bandwidth available, it is possible to assign a unique EIB channel per task, greatly reducing the risk of bus saturation. However, as the number of processors *per task* grows, presumably, the number of EIBs might also grow. However, if n (the directory size) is chosen carefully, an EIB should not happen very often. Gupta and Weber studied invalidation patterns for five applications [Gup92]. Using a 32-processor configuration, they found:

(a) For all five applications, less than 30% of the invalidations in the system affected more than one processor.

(b) For two of the applications, less than 5% of the invalidations affected more than one processor.

(c) Of the five applications, the highest percentage of invalidations affecting more than two processors was 19%.

(d) For four of the five applications, the percentage of invalidations affecting more than two processors never exceeded 8%.

4.9

(e) Of the five applications, the largest average number of processors affected by an invalidation was 1.6.

Consider a multiple channel architecture configured such that its processors are tuned to 64 channels, with a directory length of 4. In this scenario, when processors tuned to 5 distinct channels frequently cache a particular data item, the entire system gets frequent invalidation messages (via EIBs), even though a majority of the system (in this example, 59 channels) do not care about the modification. However, Gupta and Weber's results indicate that an overflow directory with a directory length of three or four will result in comparatively few global notifications (see Table 4.3). LocusRoute (a VLSI routing program) is the only application of the five they studied where such a scenario might occur relatively frequently. The global notification idea would actually work quite well for PTHOR, where most invalidations affecting more than three processors actually require global, or near-global notification.

Program	Number of Processors Affected By Invalidation						
	0 or 1	2 or 3	4 - 7	8 - 15	16 - 24	25 - 30	31
Maxflow	85.3%	10.0%	3.0%	0.8%	0.0%	0.0%	0.1%
MP3D	98.0%	1.2%	0.3%	0.2%	0.0%	0.0%	0.0%
Water	99.0%	0.0%	0.0%	0.1%	1.0%	0.0%	0.0%
PTHOR	89.0%	6.0%	0.8%	0.6%	0.1%	0.7%	2.0%
LocusRoute	70.0%	18.0%	11.0%	1.2%	0.0%	0.0%	0.0%
(Mean)	88.3%	7.0%	3.0%	0.6%	0.2%	0.1%	0.4%

Table 4.3: Processors Affected By Invalidation Write. Percentages show the percentage of invalidations which affect the said number of processors. Number of Processors Affected By Invalidation. Data shown is for 32-processor configuration. Gupta and Weber only rounded results less than 1% to the nearest tenth; results over 1% were rounded to nearest whole per cent. Hence, some totals do not add to 100%.

Because the MCA directory is channel-based, the invalidation patterns found by Gupta and Weber actually represent a worst-case. Since more than one processor may be tuned to the same channel, an invalidation which affects three processors in Gupta and Weber's analysis may actually only affect one or two channels in an MCA configuration. This means that the MCA has three advantages over conventional architectures when implementing an overflow protocol. The first advantage is *less global notifications*. A channel-based directory will fill more slowly than a processor-based directory, since any number of processors tuned to the same channel all require one and only one directory entry. A directory which fills more slowly will not overflow as often, thereby avoiding a more costly secondary notification scheme. Second, an MCA will generate *less network traffic* when the directory is not full. Whenever two processors tuned to the same channel have cached the same item, only one message is needed to notify them both of the change. Since the MCA can be configured so that processors working on the same part of a problem can be tuned to the same channel, such a savings could be realized quite often. Third, because the MCA exploits the advantages of fiber optic communications, the EIB channel uses the same "wire" as the other network traffic, without colliding with the existing network packets. This means that the MCA inherently provides a very efficient and timely method of secondary notifications.

4.5. Other Design Decisions for the MCA Overflow Protocol

Once the overflow scheme had been chosen, two design issues needed to be addressed. First, should notifications involve updates, invalidations, or a combination of both? Second, how should the directory be maintained?

These two questions are related. If invalidations are used, then the directory is self-maintained. Whenever a cache block is invalidated, that entry can be deleted from the directory. However, if updates are always used, then how do you prevent the directory from filling up with pointers to processors which no longer have the block?

Gupta and Weber's research [Gup92] was again used to answer these questions. One reason that such a dominant percentage of invalidations affected only one processor is because so much of the shared data in parallel programs is *migratory*. Migratory data is data which is typically passed from one processor to another, traversing the system, or a portion of the system.

Therefore, the MCA protocol works as follows: If there are any open directory slots, there is no imminent danger of filling the directory and forcing a global notification. Therefore, when the directory is not overflowing, updates can be sent across the channels which are in the directory, so that each processor caching the data block can receive the current value. Thus, the cached data remains valid. When the directory is overflowing, then a global notification occurs. Every channel will receive a message invalidating the data, via the Extended Invalidation Broadcast. However, the one processor which performed the write will ignore the message (allowing it to retain its valid copy), and the directory will retain that channel number in the directory. The remainder of the directory is cleared.

Two other options were considered. First, invalidations could be used, rather than updates, when the directory is not full. In this scenario, the directory is cleared of all channels except the one used by the writing processor. All other caches invalidate the block. A possible disadvantage of this scheme is that a cache may have to re-fetch a block, whereas updates keep

4.12

the block valid. The second option calls for the processor, when acknowledging receipt of the update packet, to inform the memory unit of whether or not that block was found in the cache. Therefore, if the block has been ejected, then that entry can be cleared from the directory. While this might work for some architectures using an overflow directory protocol, it was regarded as too complicated for the MCA since memory would need to make sure every processor tuned to a particular channel no longer has the block.

4.6. Storage Requirements

Let n denote the directory length (the number of entries in the limited directory), and N denote the number of channels used by the processors on a system. Every data block in memory that could potenti;1y be shared requires an associated directory with enough bits to store n channels, plus one overflow bit. n extra bits will be required indicating whether or not each channel in the directory is indeed valid. There is also one extra bit required for signaling an update-in-progress; the justification for this bit is presented in Section 5.7. So, if N channels are used by the processors, and n is the length of the directory, then the total storage requirement is:

(no. of directories) * (n * (no. of bits/channel + 2)) bits,

or,

(MEMORY SIZE words / SHARED BLOCK SIZE words) * n * $(\log_2 N + 2)$ bits.

Chapter 5 describes the possible states of the shared cache. Because there are four states, two status bits are needed per shared cache block. In the

4.13

simulator, three boolean variables are used, but the primary motivation for this configuration was software readability.

4.7. Logic

When a processor wants to modify a valid block in its cache (i.e., has a WRITE HIT), it updates the cache block, and sends the new value to main memory. When memory receives the packet, the following actions transpire, depending upon the status of the directory:

THE OVERFLOW BIT IS NOT SET: The new value of the modified word is sent over each valid channel in the directory. If a snooping cache on that channel has the block, the new value is written into the cache.

THE OVERFLOW BIT IS SET: An EIB occurs. All processors receive the address of the block to be invalidated. The processors which have cached the block invalidate that block. The one exception is the processor which performed the write—its block is valid, and the state of this block enables the processor to retain its valid copy. Only the channel of the processor which sent the updated block (the processor which retains its valid block) remains in the directory.

On a WRITE MISS, the new value is sent to memory. If the channel of the writing processor is already listed in the directory, then updates are used to keep the data coherent. If the channel is not already in the directory, then it is added (if there is room), and update messages are sent to all channels except the new one. If there is no room to add the new channel, then an EIB is performed. Figures 4.1 (a)-(c), shown at the end of this chapter, depict this behavior.

4.7. Summary

This chapter introduced an overflow directory scheme for cache coherency. It was also shown that this protocol and the MCA are very

4.14

compatible, since the channel-based directory will help eliminate some of the undesirable characteristics of an overflow directory.

Although there are many different ways an overflow directory could be implemented, some design choices were made which will be analyzed using the MCA simulator. In particular, the MCA protocol will be tested with varying directory lengths. Based on [Gup92], the expected optimum directory size is ($n = 3$) or ($n = 4$).

When a cached variable is modified, the updated block is transmitted to main memory. Main memory must acknowledge the update, in case multiple processors try to update at the same time. A second processor requesting an update is told to wait if the previous update has not completed.

If the directory for a modified block has not overflowed, updates will be sent to the caches with that block. The memory node owning the block will transmit the new value across each channel in the directory, including the channel from which the update originated (in case multiple processors on that channel have that variable cached).

If the overflow bit is set, then an Extended Invalidation Broadcast (EIB) occurs. Invalidation messages are sent to every processor assigned to this task over the EIB channel. However, the writing cache retains its valid copy, thanks to a special state which allows it to ignore this particular invalidation message. EIBs are projected to occur infrequently.

Is this coherency scheme scalable? The storage requirements do not increase as processors are added to the system. Because of the invalidation behavior of many parallel programs [Gup92], the directory size can remain small. The overflow bit allows all of the processors to cache the same variable, so that some of the performance problems associated with limited

4.15

directories can be overcome. The EIB channel provides a secondary notification scheme which is not too cumbersome.
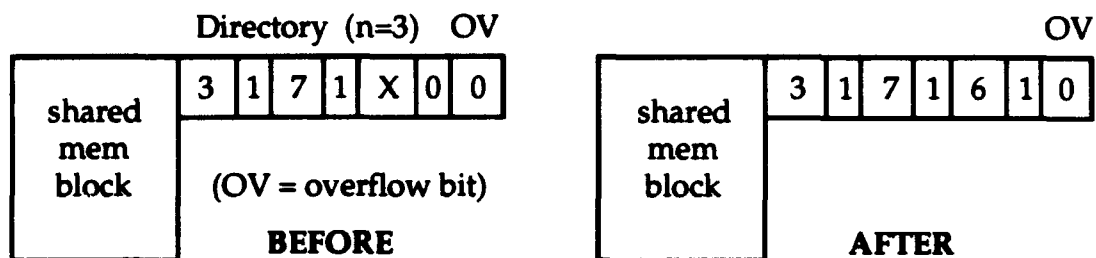
Directory (n=3)   OV

shared mem block	3	1	7	1	X	0	0

(OV = overflow bit)

BEFORE

OV

shared mem block	3	1	7	1	6	1	0

AFTER

Figure 4.1 (a). A processor on channel 6 has a WRITE MISS. Updates are sent over channels 3 and 7; channel 6 is added to the directory.

Directory (n=3)   OV

shared mem block	3	1	7	1	6	1	1

(OV = overflow bit)

BEFORE

OV

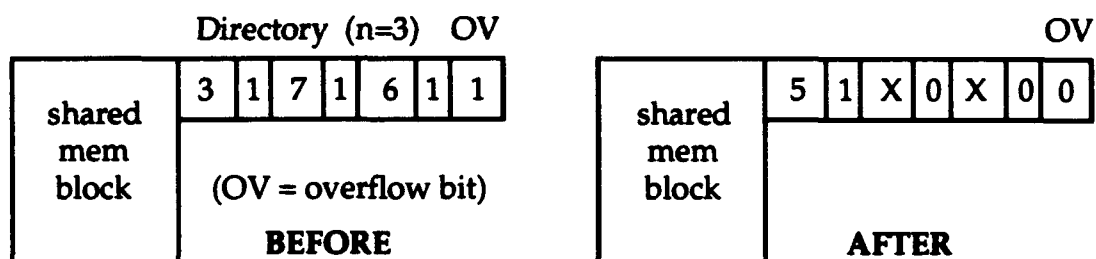| shared mem block	5	1	X	0	X	0	0

AFTER

Figure 4.1 (b). A processor on another channel (let's say channel 5) has had a READ MISS, which caused the directory to overflow (BEFORE configuration). When this processor has a subsequent WRITE HIT, an EIB is performed. Whenever an EIB is performed, the resulting directory is cleared of all entries except one. It contains the channel of the processor which wrote to the block (in this example, channel 5).

Directory (n=3)   OV

shared mem block	5	1	4	1	3	1	0

(OV = overflow bit)

BEFORE

OV

5	1	4	1	3	1	1	(i)

shared mem block

2	1	X	0	X	0	0	(ii)

AFTER
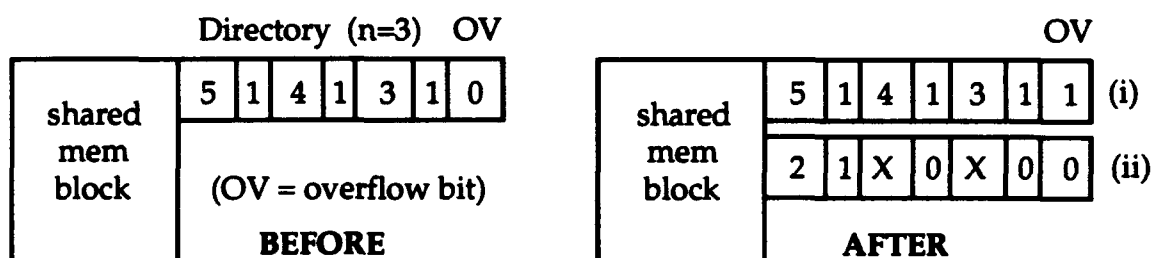
Figure 4.1 (c). When a processor which is not in the directory has a WRITE MISS, and the directory is full but not overflowing, there are two options. Assume the writing processor is tuned to channel 2. The first option is to send updates to all the channels in the directory, and then overflow (option (i)). The alternative is to perform an EIB and clear the directory (option (ii)). The MCA simulator implements option (ii).

Figure 4.1: Directory Maintenance for the MCA Protocol.

CHAPTER 5

SETTING UP THE EXPERIMENT

5.1. Implementing Cache Coherency on the MCA Simulator

A brief description of the MCA simulator was given in Chapter 3.
Chapter 4 described the proposed coherency protocol for the MCA. This
chapter describes how the MCA simulator was adapted in order to test the
proposed coherency protocol and the goals of the experiment.

5.2. The Cache Data Structure

The existing simulator defined a data structure to be used as the private
cache for each CPU. This data structure includes enough space for tag bits,
status bits, and the cached data. Because the proposed protocol separates
shared (global) and private cached data, a similar data structure, the shared
cache, was defined for each CPU. One of the advantages for separating shared
data into its own cache is that the supporting hardware can be customized for
each environment. In the MCA, private data uses a write-back scheme, that
is, modified data is written back to memory only when a dirty block is ejected.
Therefore, a dirty bit is associated with each block in the cache. However, the
shared cache uses write-through, so this dirty bit is not needed.

The private cache uses two state bits: one bit determines validity (all
blocks are invalid at start-up time); the other bit keeps track of cleanliness
(dirty blocks are written back to memory at ejection time and at task
completion). The shared cache uses three state variables: one is a validity bit
(again, all blocks are invalid at start-up time); one bit is for nonreadable
reservations, explained in the Section 5.4; the third bit is used to signify a

load-wait, which is also explained in Section 5.4. As noted in the previous chapter, only two bits are required in hardware, since there are only four possible cache block states. However, three variables were used in the simulator for the sake of clarity.

5.3. Implementing the Extended Invalidation Broadcast (EIB)

EIBs are performed on the barrier channel, and are designed to inform all the processors assigned to a particular task of a cache block invalidation. In a global broadcast, the invalidation message is sent to a large number of nodes (possibly 64 or more), so it is imperative for the entire EIB to transpire quickly. Otherwise, global broadcasts will cripple the system, and the coherency protocol will not be scalable. Sending the invalidation is not the problem, because only one transmission is required. However, for the system to be reliable, each processor must acknowledge that they received the EIB message. The question is: "How can 64 (or more) processors quickly acknowledge the receipt of this critical information?"

To solve this problem, the barrier channel is unlike the other channels on the system in that it is a time-division multiplexed (TDM) optical channel. Each frame on this channel is divided into slices: one slice per processor, and one slice for the node which initiates the event (see Figure 5.1). The initiator, a shared memory node sends the address of the block to be invalidated, and the processors all respond during their designated time slice. To date, the barrier channel is used exclusively for EIBs; however, future versions of the MCA may also exploit this capability as a faster way to pass over program barriers.
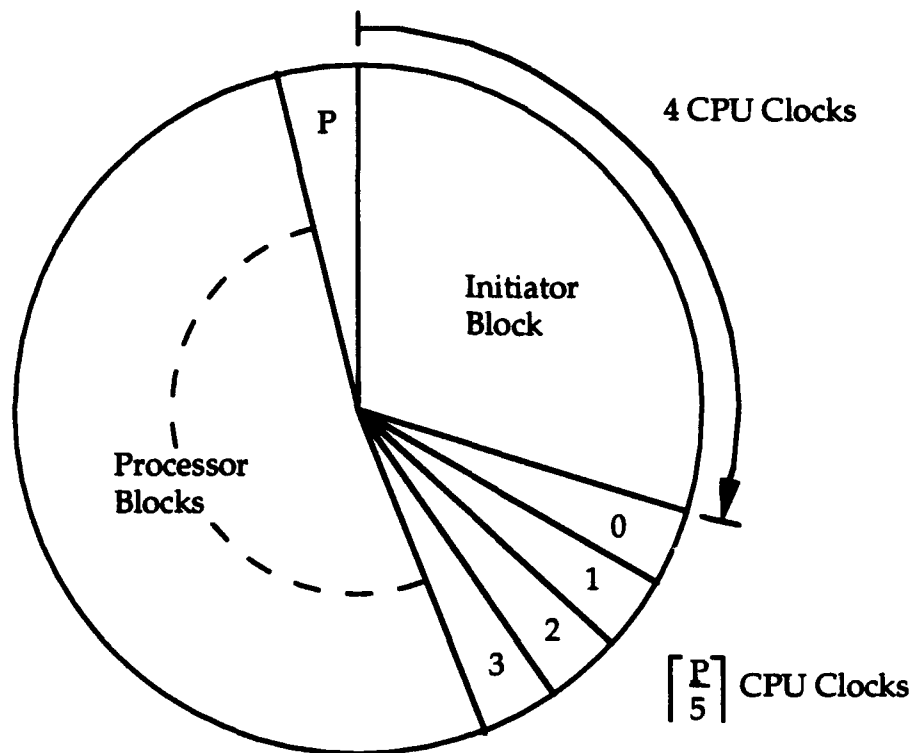
5.2

Figure 5.1: TDM Time Slice Mapping. In an MCA configuration with p processors, one time slice is dedicated to each processor. Since each processor is only required to transmit one byte, the time slice allotted for a processor is smaller than the time allocated for the initiator, which must be able to transmit the address of the block. The length of the frame in CPU clocks was calculated based on the following assumptions:

2 GBit/sec channel
50 MHz CPU clock (20 ns / CPU cycle)
1 byte reply per processor
PLAs at receiving end with 10 nsec logic delay
15 byte initiator message (see below)

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14
SYNC.		LENG.		TY	ADDRESS								CRC	

INITIATOR BLOCK: 15 bytes long. Each byte can be transmitted in 4 nsec, Therefore, this block takes 60 nsec, or three CPU clocks. Adding a 10 nsec logic delay increases the initiator block length to four CPU clocks.

KEY: SYNC = Synchronization Field
       LENG = Packet Length
       TY = Packet Type
       CRC = Cyclic Redundancy Check

5.3

There is more than one way to actually implement an optical TDM channel. Detailed discussion of possible implementations is beyond the scope of this thesis. However, [For93] describes one very appealing possibility. The design in [For93] allows multiple registers to be strobed into a buffer which is controlled by an optical clock. If adapted for the MCA, each processor's barrier r/t would be controlled in this fashion. Memory would initiate the event by tuning to the EIB channel and transmitting the block address at the start of the frame. Each processor would then transmit on this channel, their timing being controlled by the strobe and the optical clock.

The three basic states of the barrier channel are IDLE, ACTIVE, and CLEAR. IDLE means that nothing is transpiring on the channel. Once an event has commenced, the channel transitions to the ACTIVE state *and remains in the ACTIVE state until all processors have acknowledged the event*. Once all processors have acknowledged the event, the channel enters into the CLEAR state, and remains in the CLEAR state until all processors have acknowledged that they have cleared. Thus, this channel can be represented by a central data structure which has a state, and an array of bits, one per processor. When the state is IDLE, all bits are at zero (0). When the state is ACTIVE, the bits are set to one as each processor acknowledges receipt of the information. When all bits are set to one (1), the state transitions to CLEAR at the beginning of the next frame, at which time each processor can reset its bit to zero. If for some reason a processor is unable to do this, then the channel remains in the CLEAR state for the duration of the frame. The channel returns to the IDLE state only at the end of the frame when all bits have been set to zero. Because each CPU only needs to send a single byte

5.4

during its frame segment, the frame time remains a relatively small number of CPU clocks. The frame cycle time is also calculated in Figure 5.1. The assumptions used to calculate this figure are actually very conservative, and represent transfer rates obtainable in 1988 with readily available components (see [Han88], [Tuc87]).

For EIBs, the initiating node is always a shared memory node which has just received an updated line from a shared cache, with the overflow bit set, making an update impossible. Since only one EIB may transpire at a time, the memory node must first wait until the EIB state is IDLE. Once it is idle, the state is changed to ACTIVE at the beginning of the next frame. The time slice allocated for the initiator is long enough for the initiator to broadcast the address of the updated line (in addition to the standard preamble necessary), *and* long enough for this information to traverse the network. Thus, in the best case, the processor assigned time slice 0 (as well as the subsequent processors) are able to acknowledge the message in the same time frame. However, acknowledgement is only allowed when the shared cache controller is FREE and can therefore check to see if the line is present in the cache. Additionally, this cache controller will not be able to process any infcrmation received from the conventional channels until after the delay associated with checking for a cache line and updating a cache line status has transpired. If the cache controller is not FREE, then the processor will try again during the next frame, and the state of the EIB channel remains ACTIVE. EIBs always have higher priority than other buffered messages, so if a processor has to buffer the EIB (due to a busy cache), then the EIB will be processed right after the current cache transaction is finished.

5.5

5.4. Non-readable Reservations in the Shared Cache

Recall that cache coherency requires that a read to any address in memory returns the last value which was written to that address. However, this is very difficult to obtain with certainty in a parallel environment, where packet collisions in the network can cause delays and out-of-order arrivals at the network nodes. The proposed MCA protocol has another consideration. Because the specialized EIB requires less overhead than general purpose messages, an EIB is significantly quicker than other traffic in the network. Therefore, an invalidation could "beat" a packet to a particular processor. Consider the following scenario:

(1) Processor A requests shared block Z from memory node M.

(2) Memory node M receives the request, assembles a packet with block Z, and puts the packet in its transmit buffer.

(3) While attempting to transmit to processor A, a collision occurs. By order of the CSMA/CD arbitration scheme, memory node M's packet waits.

(4) Processor B writes to block Z, and sends the new line to memory node M.

(5) Memory node M receives the new line from processor B. Assume that the directory for block Z is overflowing. An EIB occurs.

(6) Processor A receives the EIB. Seeing that the line is not in its cache, it sets its barrier bit but ignores the invalidation.

(7) Processor A finally receives its packet from memory node M. *This packet contains stale information.* The data is no longer coherent.

In order to prevent this from happening, the MCA uses a *reservation required* scheme. It works as follows: whenever a shared block is requested, the block's address is put into the cache *before the request is sent to main*

memory. This is called the *reservation.* The state of the reserved block is initialized to VALID but NOT READABLE due to LOAD-WAIT.

A valid reservation is required in order to load a block of memory into the cache. When a requested block is received from main memory, it can only be put into the cache in a VALID spot, with the correct block address already in the cache. If such a reservation is found, the data is loaded into the cache, and the state of the block is changed to VALID, READABLE. Otherwise, the reservation was canceled by an EIB. When an EIB is received for any VALID block, the state is changed to INVALID, regardless of the status of the READABLE bit. When the reservation is canceled (not found), a new reservation is made, and the cache requests the block again.

5.5. Simultaneous Reads and Writes—Another Coherency Problem

Suppose the MCA is configured with a directory of length three (3), and that this directory is not overflowing. Furthermore, there are two processors assigned to each channel. Three processors, on three different channels, have cached the data. Two of them are performing a sequence of reads, and the third performs a write. The following events must transpire:

(1) The third processor assembles a packet, and transmits to memory (because this could cause a collision, it may be quite a few clocks before the packet is even put on to the bus).

(2) Memory receives the packet, forms a packet of its own with the new line, and must transmit this packet over *three* different channels (let's assume that two packets collide and the third is successfully transmitted on the first try).

(3) The three processors eventually receive the update packets, *at different times, while two of them are performing a series of reads to the block.*

(4) Six processors send acknowledgement packets back to memory.

5.7

Clearly, it will be difficult in such a scenario to guarantee coherency, that is, every read performed returns the last value written to memory. The question is, at what point in time does the write take place? The scenario above may take over 100 CPU clocks to complete. A policy must be developed which reconciles the narrow definition of cache coherency with the timing issues of the parallel environment. The MCA uses a policy which states that the writing cache is the last processor able to use a modified block, and that it can do so only after all other processors have received and acknowledged the update/invalidation. Thus, the write is not considered complete until the writing processor has received a write-grant from memory. In the scenario above, the writing cache sets its NOT READABLE bit before step one, and it is not allowed to use the modified block until the bit is reset (the bit is reset after the processor receives the *write-grant* message from memory). This procedure defines the exact time when a write has taken place, *after the memory sends a write-grant to the processor*. This implies that in order to ensure that a processor never accesses the written value too late (i.e., never reads a stale value), it will sometimes receive a new value "early". That is, it may read the updated value before the write is considered complete. What is guaranteed, however, is that when a processor writes to memory, no other processor will read a stale value while the write is in progress, because the written value is not used by the writing processor until after the coherency actions have completed.

5.6. Summary of Shared Cache State Bits

The meaning and use of the state bits in the shared cache can be summarized as follows:

(1) On a READ MISS, a reservation is made, and the bits are set to VALID, NOT READABLE, LOAD-WAIT. When the requested line arrives from memory, the status is changed to VALID, READABLE, and the LOAD-WAIT bit is reset. If, on the other hand, an EIB is received before the line, the status of the line becomes INVALID, which will force a subsequent reservation and request when the line arrives from memory.

(2) On a WRITE HIT, the line is set to VALID, NOT READABLE. If an EIB is received with this configuration, *the status remains unchanged*. This configuration allows the writing processor to retain its valid copy of the block. The NOT READABLE bit is reset after the write-grant message is received.

(3) On a WRITE MISS, a VALID, NOT READABLE reservation is made. Is the same as a WRITE HIT, except that memory will send the line back with the write-grant request, at which time the status is changed to VALID, READABLE.

Figure 5.2 depicts a state diagram showing the possible states and the transitions between these states.

5.7. Simultaneous Writes--How the Protocol Handles Them

When two writes occur at the same time, or near the same time, and the values differ, only one value can be used. But which one? Because the processors are independent, and messages must pass through a network, there is no real way to determine which processor was "first" and which was "second." Additionally, two simultaneous writes could "confuse" the protocol, if two EIBs were to transpire simultaneously. To prevent this confusion, each directory has an EIB/Update-in-Progress (EIP) bit. This bit prevents any request from being processed until the coherency actions from a previous write have completed.
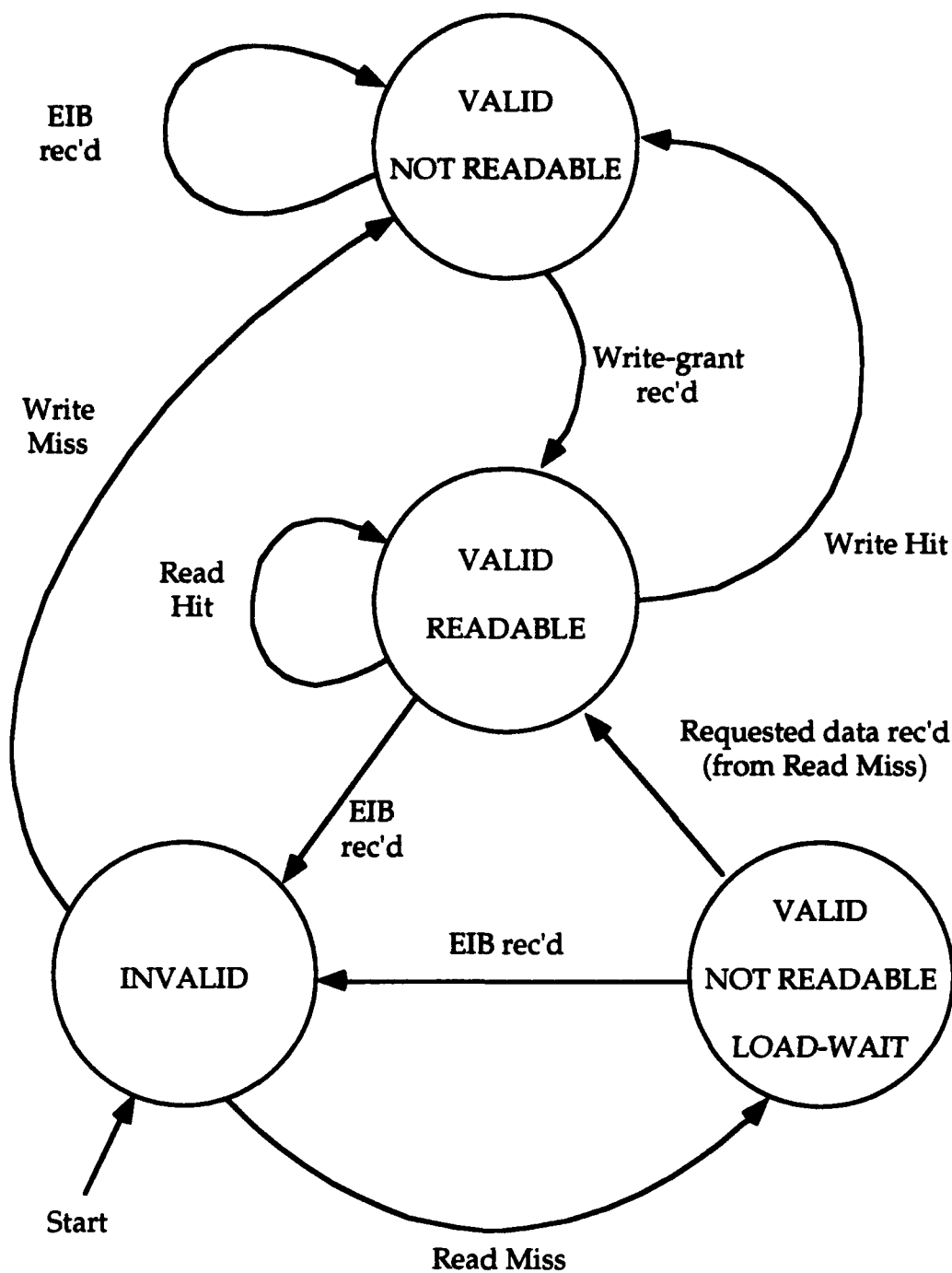
Figure 5.2: Shared Cache Block State Diagram for the MCA Coherency Protocol.

5.10

The following scenario describes how the MCA enforces coherency in the simultaneous write situation:

(1) Processors A and B write different values to address Z. Both have cached the block, so each processor has a write hit. They both change the state to VALID, NOT READABLE, and send their respective modified blocks to memory M.

(2) Memory M will receive one of modified blocks first. Without loss of generalization, assume it is the packet from processor A. If an EIB occurs, both processors, A and B, receive notice of an invalidation. Since each one knows it has modified that block (the NOT READABLE bit is set), neither invalidates. In an update, both caches will put processor A's version of the block into their lines. In either case (EIB or update), the blocks stay unreadable.

(3) The packet from processor B is received by memory M. Memory will buffer this packet until the update from processor A has been *completely* handled, and the EIP bit is reset.

(4) Processor A is sent a write-grant, after (a) the EIB is complete, or (b) the receipt of all update packets has been acknowledged. After processor A acknowledges the write-grant, the EIP bit is reset. Then processor B's request is read from the buffer. *This request (a WRITE) will cause either an EIB or an update.*

(5) If an EIB is performed, then processor A will now invalidate, since its NOT READABLE bit is no longer set. Processor B will ignore this EIB, as it did the first. If an update is used, both caches will receive a new value from memory, and update their caches. But initially, only processor A will be able to use the value at this time.

(6) Processor B will get a write-grant from memory. The net result, regardless of whether EIBs or updates are used, is that memory M contains the second value received for address Z, and the processor which gets the last write-grant has a valid block in its cache.

5.8. The Experiment

The MCA simulator was adapted to include the coherency protocol as described above. Next, the modified simulator was used to collect output from several configurations in hopes of answering pertinent questions.

5.11

Some of these questions are listed below:

(1) What performance gains are realized when the shared coherent caching scheme is used? How does this vary as the tasks and configurations are more demanding?

(2) What size directory (directory length) yields optimum performance? Does the optimum length change as the number of processors increases? What affect does increased load on processor buses have on the optimum directory size?

(3) What are the optimal shared cache parameters (block size, overall size, associativity)? Do these differ from the private cache parameters? The previously existing simulator used a 64K, 4-way set associative, 32-byte block size private cache; [Wai92] asserts that these parameters are near-optimal.

The purpose of these tests was not to prove the architecture itself; the architecture was proven in extensive testing done in [Wai92]. Rather, the purpose was to measure the effectiveness of the coherency scheme and the performance benefits realized by caching shared data.

In order to answer these questions, configurations of 4, 8, 16, 32, 64 and 128 processors were used. The configuration with 4 processors had no bus sharing among processors, and bus sharing gradually increased as the number of processors grew. The specific configurations are shown in Table 5.2. For each configuration, baseline tests were run, using the previous version of the simulator. The baseline test results (which are used for comparison purposes in Chapter 6) are shown in Table 5.1.

The first experiment goal was to find an optimum directory size, and to see if this size varied according to the number of processors on the system. This was done by running the various configurations with a 64K private cache and a 32K shared cache, and varying the directory size from zero to four.

Program Name	Number of CPUs	Number of Insts[1]	Number of CPU Clocks[1]	Percent Utilization
MATRIXMULT	4	18.4	38.9	87.2
(128 x 128)	8	9.4	20.2	85.6
	16	4.9	10.9	82.2
	32	2.7	6.5	74.5
	64	1.5	4.9	56.2
	128	1.0	5.7	30.1
FILTER	4	2.21	4.79	81.4
(64 x 64)	8	1.21	3.03	70.6
	16	0.71	2.15	58.9
	32	0.46	1.66	49.7
	64	0.34	1.45	41.6
	128	0.28	1.26	39.3

Table 5.1: Baseline Simulation Results. Baseline Results (System Performance Using the Previously Existing Version of the MCA Simulator) for the Six MCA Configurations. Data shown is for the master processor, which executes the most instructions on the run. Percent utilization and instruction count for other processors is lower, since these processors spend more time at barriers waiting for the master processor. Therefore, the master processor is the critical one in terms of performance.

([1] in millions)

Once the optimum directory size was found, this was used to measure the improved performance realized by caching the shared data.

Finally, varying the shared cache parameters could help determine whether or not a dissimilar private and shared cache set-up was indeed a valid benefit gained by separating the caches. The parameter of primary interest is line (block) size, but other variables tested include associativity and overall size.

4-Processor Configuration

4 Processor Nodes	4 Processor Buses (1 CPU per Bus)
4 Shared Memory Nodes	4 Memory Buses
4 Private Memory Nodes	(1 Shared Mem & 1 Private Mem per Bus)

8-Processor Configuration

8 Processor Nodes	4 Processor Buses (2 CPUs per Bus)
4 Shared Memory Nodes	4 Memory Buses
8 Private Memory Nodes	(1 Shared Mem & 2 Private Mems per Bus)

16-Processor Configuration

16 Processor Nodes	4 Processor Buses (4 CPUs per Bus)
8 Shared Memory Nodes	4 Shared Mem Buses (2 Shared Mems per Bus)
16 Private Memory Nodes	4 Private Mem Buses (4 Priv. Mems per Bus)

32-Processor Configuration

32 Processor Nodes	8 Processor Buses (4 CPUs per Bus)
32 Shared Memory Nodes	8 Shared Mem Buses (4 Shared Mems per Bus)
32 Private Memory Nodes	8 Private Mem Buses (4 Priv. Mems per Bus)

64-Processor Configuration

64 Processor Nodes	8 Processor Buses (8 CPUs per Bus)
32 Shared Memory Nodes	8 Shared Mem Buses (8 Shared Mems per Bus)
64 Private Memory Nodes	8 Private Mem Buses (8 Priv. Mems per Bus)

128-Processor Configuration

128 Processor Nodes	16 Processor Buses (8 CPUs per Bus)
64 Shared Memory Nodes	16 Shared Mem Buses (4 Shared Mems per Bus)
128 Private Memory Nodes	16 Private Mem Buses (8 Priv. Mems per Bus)

Table 5.2: Six MCA Configurations Used for Testing.

5.9. Summary

This chapter explained in detail how some the final design decisions for the MCA coherency protocol were incorporated into the simulator. It then listed a set of questions which the numerous simulator runs were intended to answer. A detailed list of these simulations, and the overall findings, are summarized in the next chapter.

CHAPTER 6

EXPERIMENT RESULTS

6.1. The Six Test Configurations

Chapter 5 (Table 5.1) listed six MCA configurations which would be used to test the simulator. Figure 5.3 listed baseline results for each configuration, using MATRIXMULT (128 x 128 matrix size) and FILTER (64 x 64 median filtering). This chapter will use an abbreviated naming convention which combines the application program with the MCA configuration upon which it is run. For example, FILTER 16 refers to the 16-processor configuration running the FILTER program; MATRIXMULT 64 refers to the 64-processor configuration multiplying the 128 x 128 matrices.

6.2. Test Plan Overview

Because the cache invalidation patterns revealed in [Gup92] were heavily used during the design of the protocol, the first step of the testing was to see if the invalidation patterns on the MCA simulator matched those in [Gup92]. The essential goal was to confirm two of their most significant findings: (1) the majority of global invalidations actually affect relatively few processors, and (2) many invalidations affect only one processor. Gupta and Weber speculate in [Gup92] that migratory data behavior largely accounts for this behavior, as was described in Section 4.5.

The next step was to determine the optimal directory length. Various configurations of the simulator were tested where only the size of the directory was changed. These tests provided data so that the relationship between directory length and overall performance could be analyzed.

6.1

After the optimum directory size was determined, this size was used for the remainder of the testing. The next step was to determine what performance gains could be realized by caching the shared (global) data. Finally, some of the shared cache parameters such as block size and overall cache size were varied to determine their effect on hit rate and overall performance.

6.3. Confirming Expected Invalidation Patterns

Chapter 4 theorized that a relatively small limited directory could sizably decrease the number of global notifications (EIBs) during a program execution, because the invalidation patterns described by Gupta and Weber [Gup92] indicate that most invalidations affect only a few processors. The results from MATRIXMULT 128, MATRIXMULT 32, MATRIXMULT 8, FILTER 64, FILTER 16, and FILTER 4 were all analyzed, and showed that the invalidation patterns roughly mirrored those found in [Gup92], thereby confirming this assumption. The data from the two largest simulations, MATRIXMULT 128 and FILTER 64, are shown in the Figures 6.1 through 6.3. Figure 6.1 shows the total number of EIBs performed during the two simulations, with directory lengths of 0, 2, and 4. A directory length of zero forces the MCA to perform an EIB for every shared write. With directories, updates can be performed over the channels listed in the directory, and EIBs occur only when the directory overflows. The most dramatic decrease occurs in FILTER. A directory of length 2 eliminates approximately 77% of the EIBs, and a directory of length 4 eliminates 90% of the EIBs. For MATRIXMULT 128, a directory of length 4 eliminates 80% of the EIBs. With a directory length of four, the EIB is not a frequent event. For MATRIXMULT 128, only

6.2

679 EIBs occur while the master processor alone executes over 1,000,000 instructions. Figures 6.2 and 6.3 show results from the two applications when the directory size was set to zero, thereby graphically showing how many processors are effected by each shared write. Figures 6.4 and 6.5 show the total number of EIBs for all six configurations (directory size zero). These two graphs also depict the fraction of the EIBs which affect only one processor. As can be seen, this fraction is a significant portion of all EIBs, supporting the premise in [Gup90] that migratory data behavior accounts for a large percentage of all invalidations. Thus, a small directory can accommodate most coherency updates and diminish the need for global notification.

6.4. The Optimum Directory Size

Much of the protocol's design was based on predicted invalidation patterns, a prediction which has been confirmed. The next task was to measure the performance as the directory size changed. The same test runs listed in Section 6.2 were used, that is, MATRIXMULT 128, MATRIXMULT 32, MATRIXMULT 8, FILTER 64, FILTER 16, and FILTER 4. All six of these simulations were run with directory lengths of 4, 3, 2, 1, and 0 (30 simulations runs total). The results were rather surprising in that, for all the applications except one, the best performance was attained using a directory size of 0. A directory size of 1 performed better in FILTER 4, but only nominally better. The following table (Table 6.1) lists the number of clocks needed to complete the program for each simulation. Figures 6.6 and 6.7 graph these results, using CPU utilization as a function of directory size.

The question is: if the invalidation patterns occurred as predicted, why does a small directory degrade performance? The reason is because the MCA

6.3

Total Number of EIB's
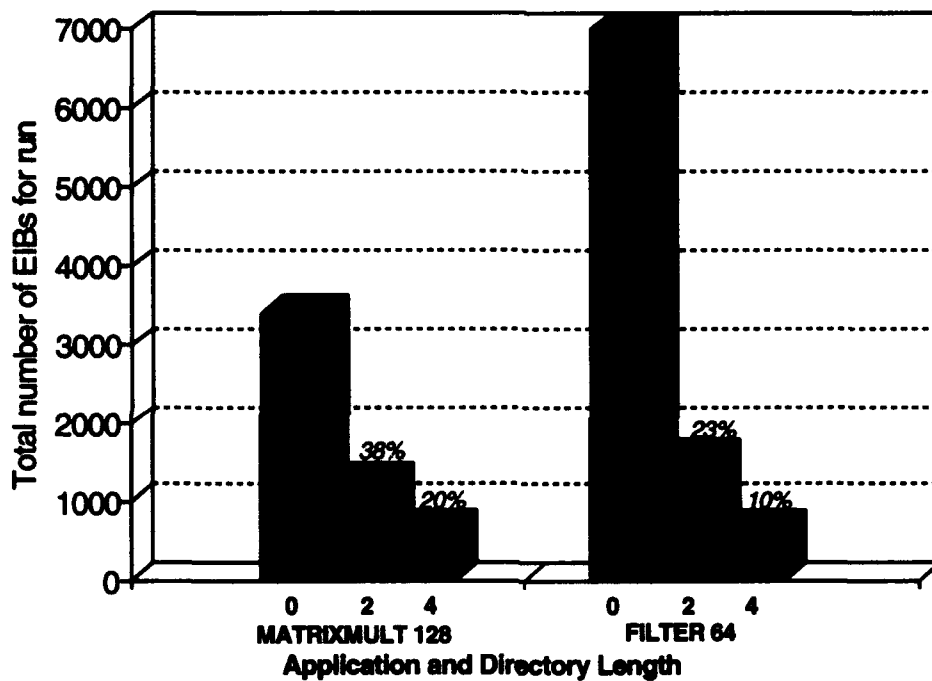### (bigger directories allow updates in lieu of EIBs)

Figure 6.1: Effect of Varying Directory Sizes on the Number of EIBs for Two of the Bigger Configurations, MATRIXMULT 128 and FILTER 64. A directory length of 4 eliminates 80% of the EIBs for MATRIXMULT 128, and 90% of the EIBs for FILTER 64.

6.4

Invalidation Pattern: MATRIXMULT 128
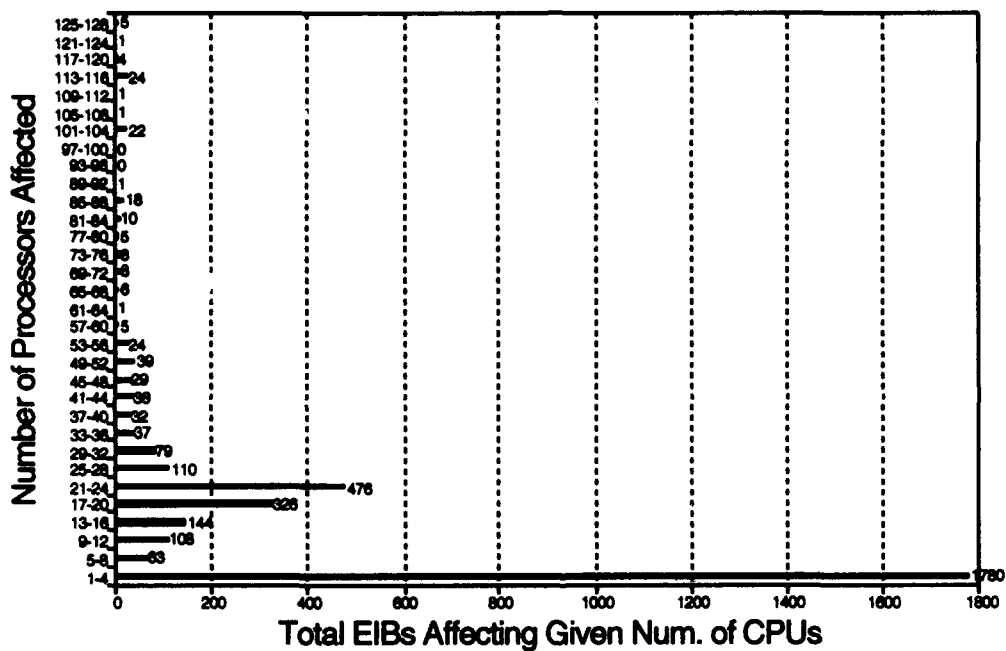### As Predicted, Most Affect only 1-4 CPUs

Figure 6.2: Cache Invalidation Pattern for MATRIXMULT 128.

Invalidation Pattern: FILTER 64
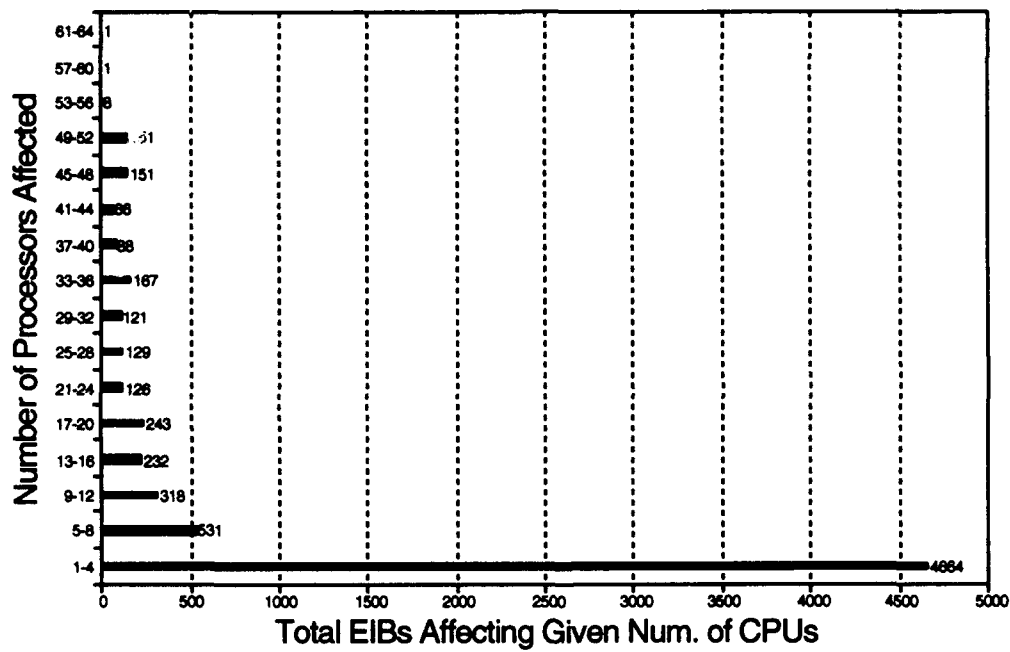Only 1/3 of EIB's Affect 5 or more CPUs
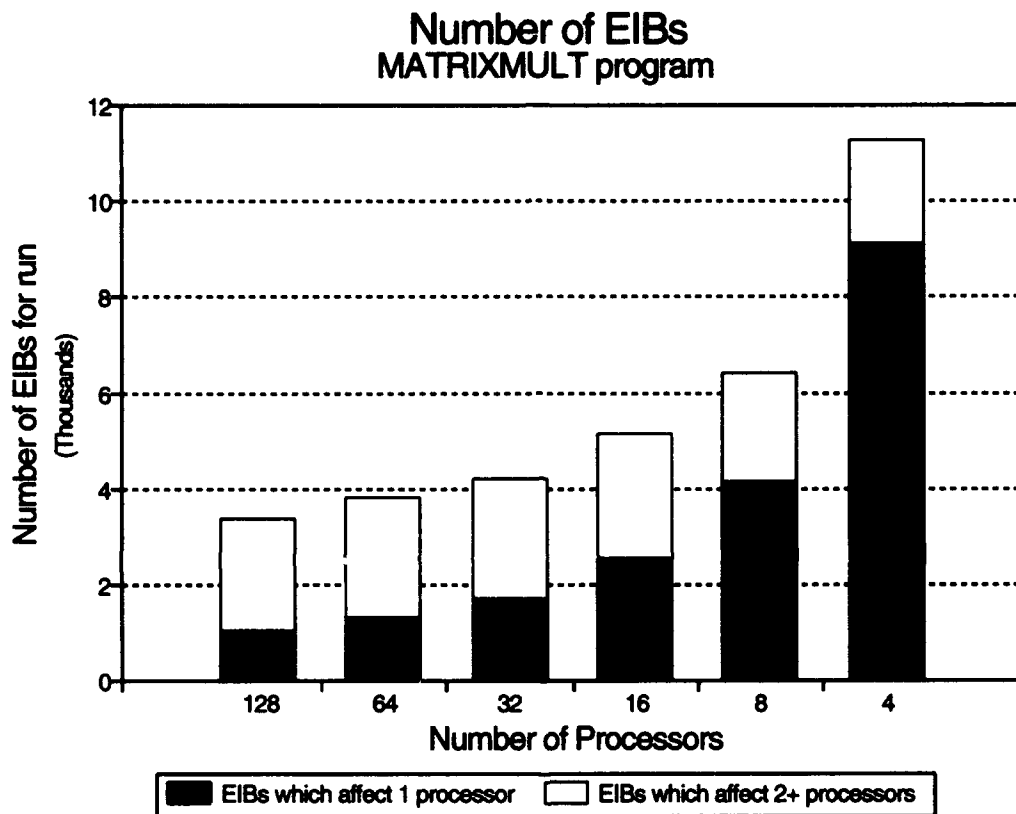
Figure 6.3: Cache Invalidation Pattern for FILTER 64.

6.6

Number of EIBs
MATRIXMULT program

Figure 6.4: Total Number of EIBs for MATRIXMULT Under Varying Configurations. The bottom portion of each bar represents the number of EIBs affecting one and only one other processor.

6.7

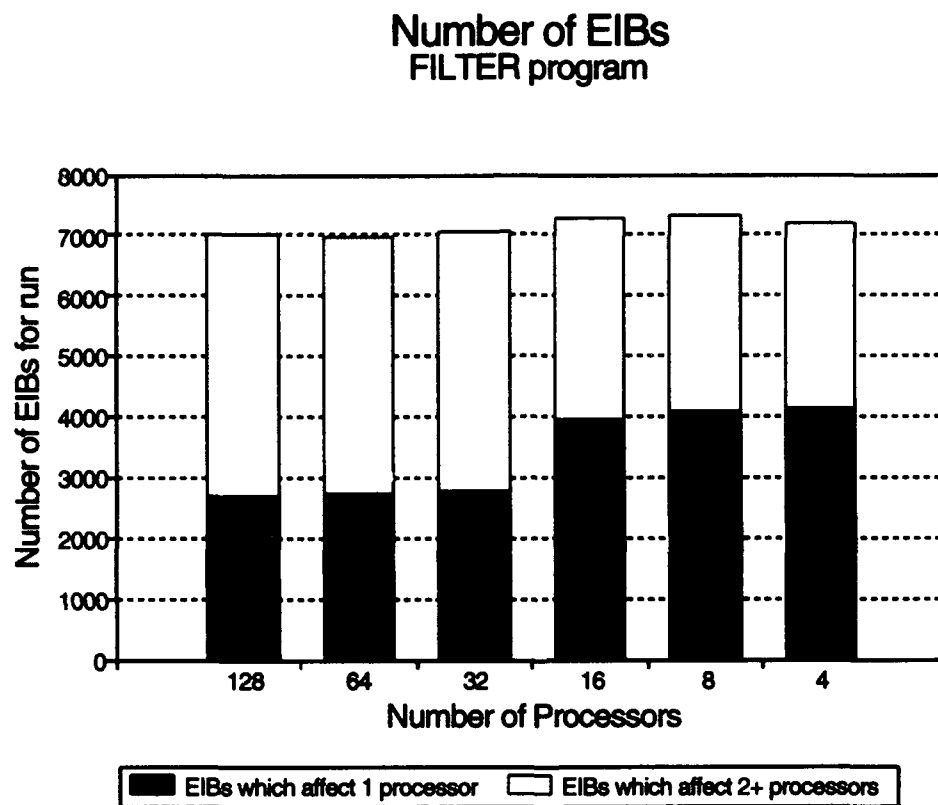# Number of EIBs
FILTER program

Figure 6.5: Total Number of EIBs for FILTER Under Varying Configurations.
The bottom portion of each bar represents the number of EIBs affecting one
and only one other processor.

6.8

DIRECTORY LENGTH =	4	3	2	1	0
APPLICATION					
FILTER 4	4.26	4.17	4.12	4.07	4.09
FILTER 16	4.87	3.18	2.54	1.92	1.55
FILTER 64	7.22	5.40	3.72	2.36	0.94
MATRIXMULT 8	21.2	20.1	19.4	19.1	9.0
MATRIXMULT 32	7.8	7.4	6.7	6.1	5.7
MATRIXMULT 128	6.2	5.4	4.3	3.4	2.5

Table 6.1: Number of Clocks (in Millions) Needed to Complete the Simulation, as the Directory Size Varies. In all cases but one, the completion time decreases as the size of the directory becomes smaller. Values in **bold** print represent performance **worse** than the baseline results, in which no shared caching was performed.

requires acknowledgement packets from the CPUs every time memory sends a packet to a processor.

Consider MATRIXMULT 8, which has two CPUs tuned to each processor channel. Assume the directory size is four, and that three of the entries are being used when a write occurs. Memory sends out three packets, which are received by six CPUs (two CPUs per channel). This process is done rather easily. However, memory must be sure all CPUs have received the update messages—this is done by acknowledgement packets. Six CPUs will all try to send acknowledgements at the same time. Since these six messages must all be sent on the same frequency (channel), collisions occur. The CPUs receive the data almost immediately, but the acknowledgement process is

Master CPU Utilization vs. Directory Size
MATRIXMULT program

Legend:
- MATRIXMULT 8 (w/ Shared Cache)
- MATRIXMULT 8 (Baseline)
- MATRIXMULT 32 (w/ Shared Cache)
- MATRIXMULT 32 (Baseline)
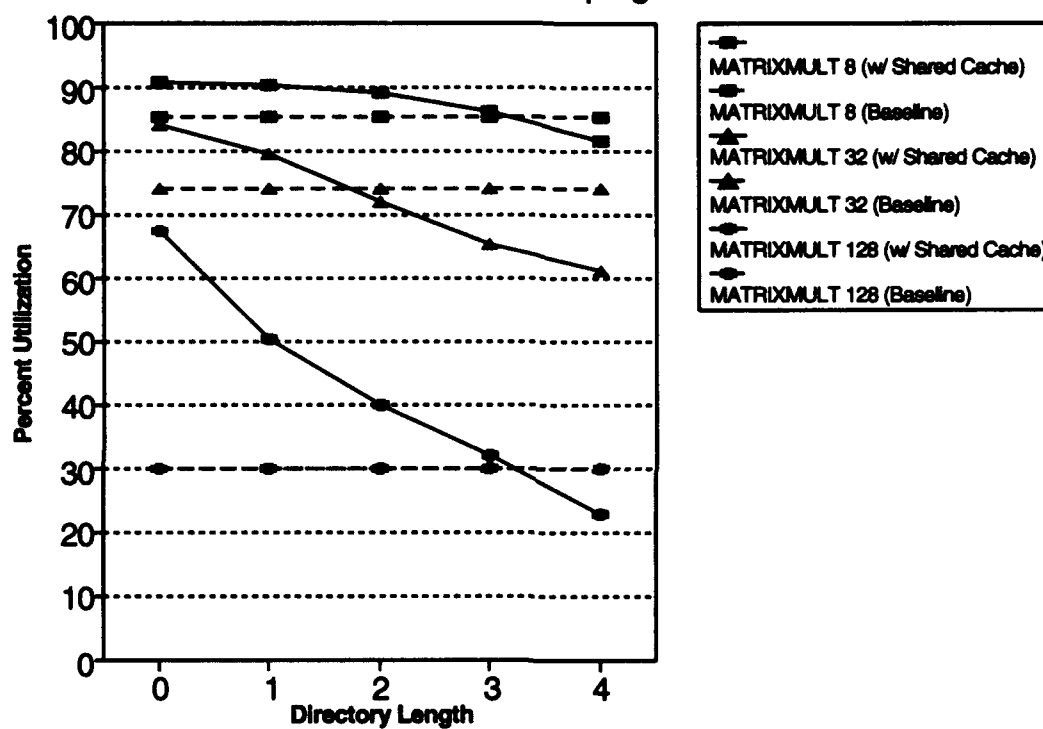- MATRIXMULT 128 (w/ Shared Cache)
- MATRIXMULT 128 (Baseline)

Figure 6.6: Percent Utilization of the Master CPU as the Directory Length Increases, for the MATRIXMULT Program. The dotted lines indicate the performance of the baseline runs.

6.10

Master CPU Utilization vs. Directory Size
FILTER program

Legend:
- FILTER 4 (w/ Shared Cache)
- FILTER 4 (Baseline)
- FILTER 16 (w/ Shared Cache)
- FILTER 16 (Baseline)
- FILTER 64 (w/ Shared Cache)
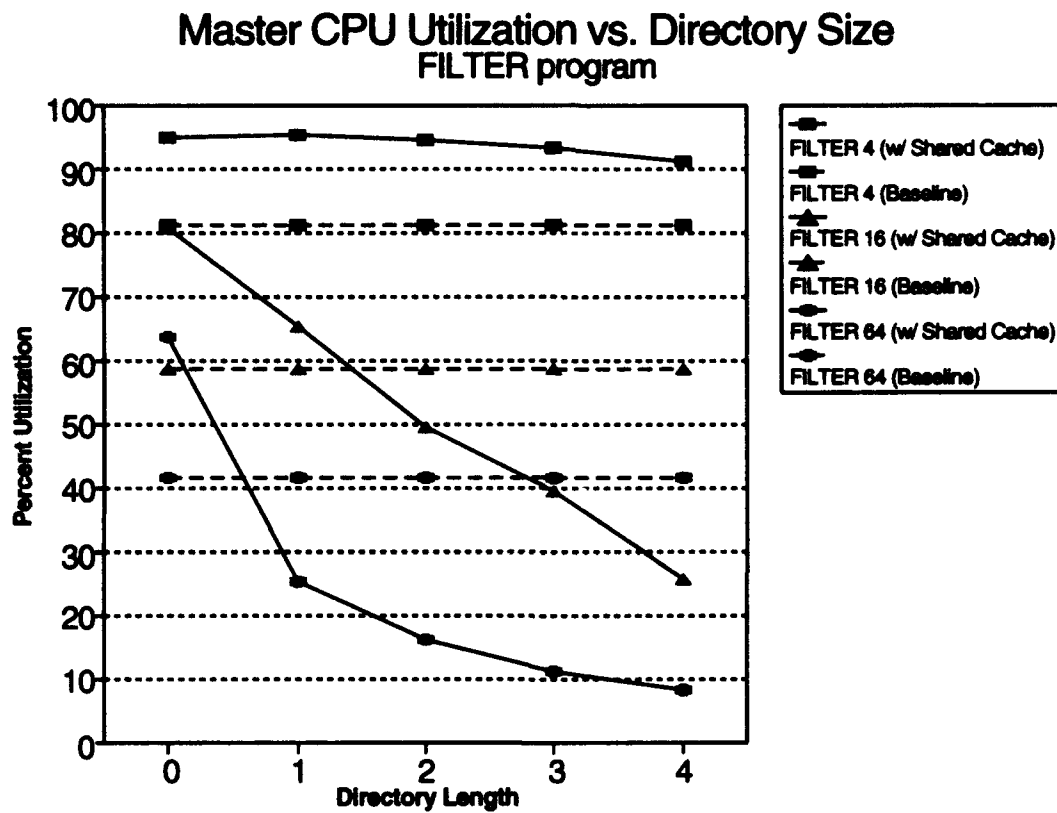- FILTER 64 (Baseline)

Figure 6.7: Percent Utilization of the Master CPU as the Directory Length Increases, for the FILTER Program. The dotted lines indicate the performance of the baseline runs.

6.11

very time-consuming, and it ties up the memory unit for a long time. Close analysis of simulation data showed that it sometimes took up to 300 clocks for all the acknowledgements to be processed by memory in this configuration. When eight processors share the same channel, and the directory size is four, up to 32 acknowledgement packets may be caught in this bottleneck.

The EIB, on the other hand, occurs on a separate channel from other network traffic, so it does not hinder normal transactions. Additionally, the special-purpose nature of the EIB channel allows the entire invalidation process to transpire quickly. Because this channel is time-division multiplexed, and each CPU's "acknowledgement" is one byte long, the message transmission time is significantly smaller than that of standard acknowledgement packets. The end result, therefore, is that secondary notifications (EIBs) are not some cumbersome procedure to be dreaded, but they are, in and of themselves, an efficient way to maintain coherency. This explains why the simulations with a directory size of zero perform so much better than the directory-based configurations. As can be seen in Figures 6.6 and 6.7, the performance decrease is more dramatic as the number of processors per channel increases. This is because more acknowledgement packets are generated per directory entry. The most dramatic example is FILTER 64 in Figure 6.7, where only the directory size of zero outperforms the baseline simulator. In other words, caching FILTER 64 with any size directory is worse than simply not caching shared data at all.

This implications of this are very important. A directory scheme which maintains a central directory at main memory, and requires acknowledgements from the processors to a central directory, is not scalable. If such a scheme is to indeed be scalable, acknowledgements must be done by

6.12

some other method. Simply eliminating acknowledgements altogether is a very questionable alternative, since a lost message could render a cache incoherent.

While the directory performed very poorly on the MCA, the EIBs performed very well. This scheme, by itself, shows more promise of being scalable. Shared cache blocks are invalidated quickly, and although the time required to perform an invalidation does increase with the number of processors, the time increase is linear.

6.5. Performance Improvement Gained by Shared Caching

Peak performance occurs with a directory of length zero. Once this conclusion was reached, the goal was to measure the improved performance realized by caching shared variables, and using this optimum configuration.

Figures 6.8 and 6.9 show the execution time of the two applications (MATRIXMULT and FILTER) expressed as a function of the number of processors used in the six test configurations. The shaded region in these two graphs shows the extra clock cycles required to finish the program when shared caching is not used. As can be seen, the shared caching always results in faster execution time.

Figure 6.10 shows the performance increase for all six configurations, expressed in terms of "X is s% faster than Y". The performance increase was measured using the following convention [Hen90]:

Let s = speedup percentage, where X is s% faster than Y.

If Ex_Y = (Execution Time of Y),
And Ex_X = (Execution Time of X),
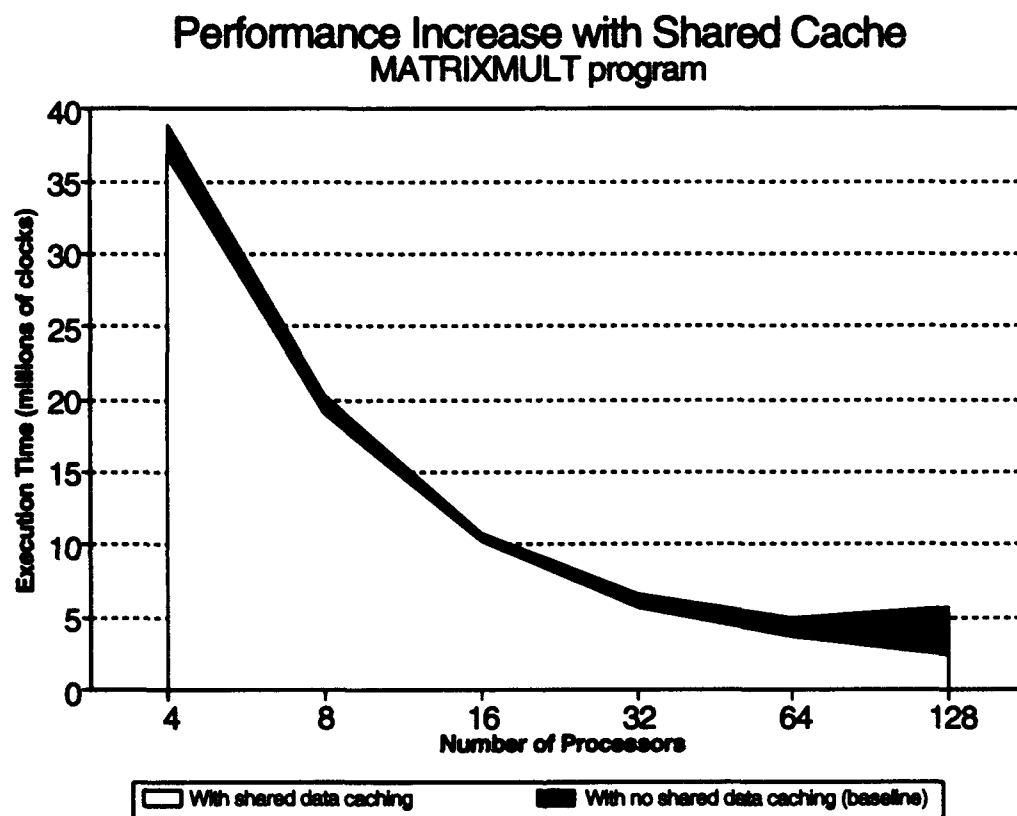Then $(Ex_Y / Ex_X) = 1 + (s / 100)$

Performance Increase with Shared Cache
MATRIXMULT program

Figure 6.8: Execution Time for Baseline MATRIXMULT, and for
MATRIXMULT with Shared Caching, Directory Size of Zero.

6.14

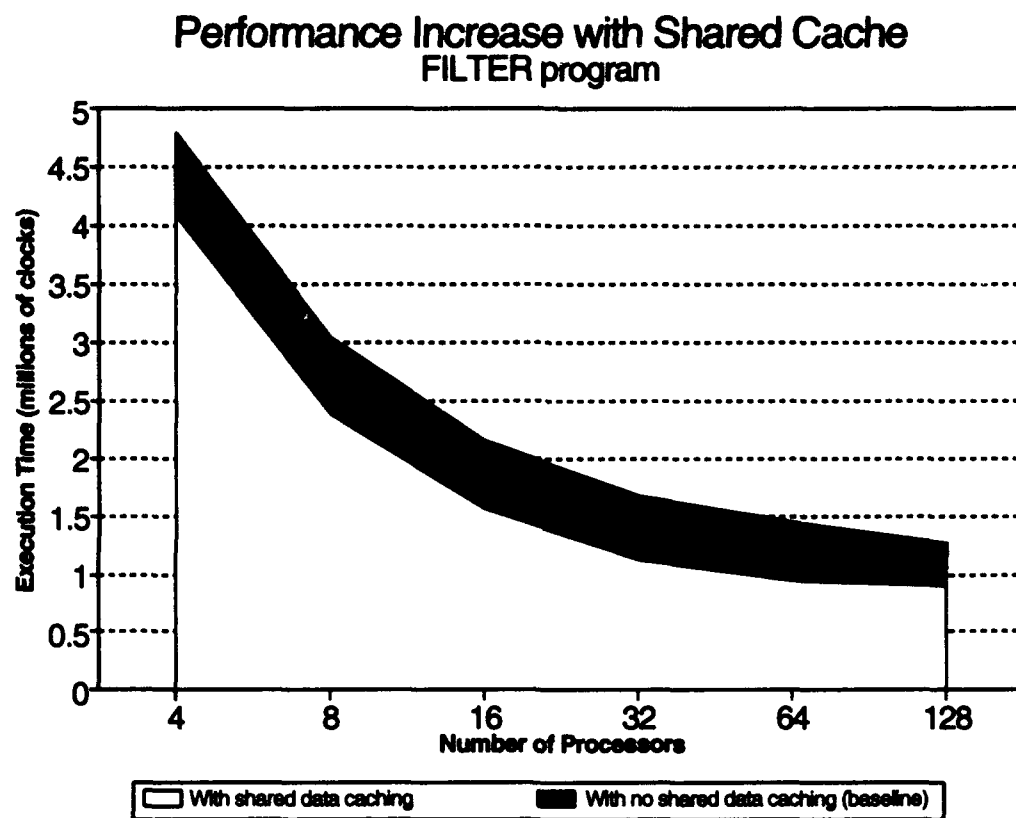**Performance Increase with Shared Cache**
FILTER program

Figure 6.9: Execution Time for Baseline FILTER, and for FILTER with Shared Caching, Directory Size of Zero.

The results show that the speedup characteristics for the two programs differ. MATRIXMULT has only a slight performance advantage for the smaller number of processors, while an enormous speedup (128 percent, or less than half of the execution time) is realized at 128 processors. FILTER, on the other hand, has a more even speedup across configurations, the peak percentage speedup occurring at 64 processors.

It is difficult to rate the speedups which are realized. It is true that the performance improved, but it is hard to know whether or not even better performance may have been achieved by using a different protocol. However, we know that a CPU can never exceed a 100% utilization. So we can use CPU utilization percentage as a basis, and safely say that the protocol is satisfactory if the utilization figures always improve. As utilization nears 100 percent, we can assert more strongly that the protocol is as effective as possible. Table 6.2 shows the percent utilization for the test and baseline runs. There is marked improvement for all six configurations, so we can conclude that the protocol performs satisfactorily, at least for the two applications tested. Of the two applications, MATRIXMULT is the more parallelizable algorithm, so it is expected to achieve higher utilizations, especially for the larger configurations. The huge leap in utilization for MATRIXMULT 128 (from 30.1% to 67.8%) is very encouraging. It is also worth noting that in the baseline simulation, the matrix multiply took longer to complete in the configuration with 128 processors than it did in the 64-processor configuration. However, with shared caching, performance continued to improve at 128 processors.
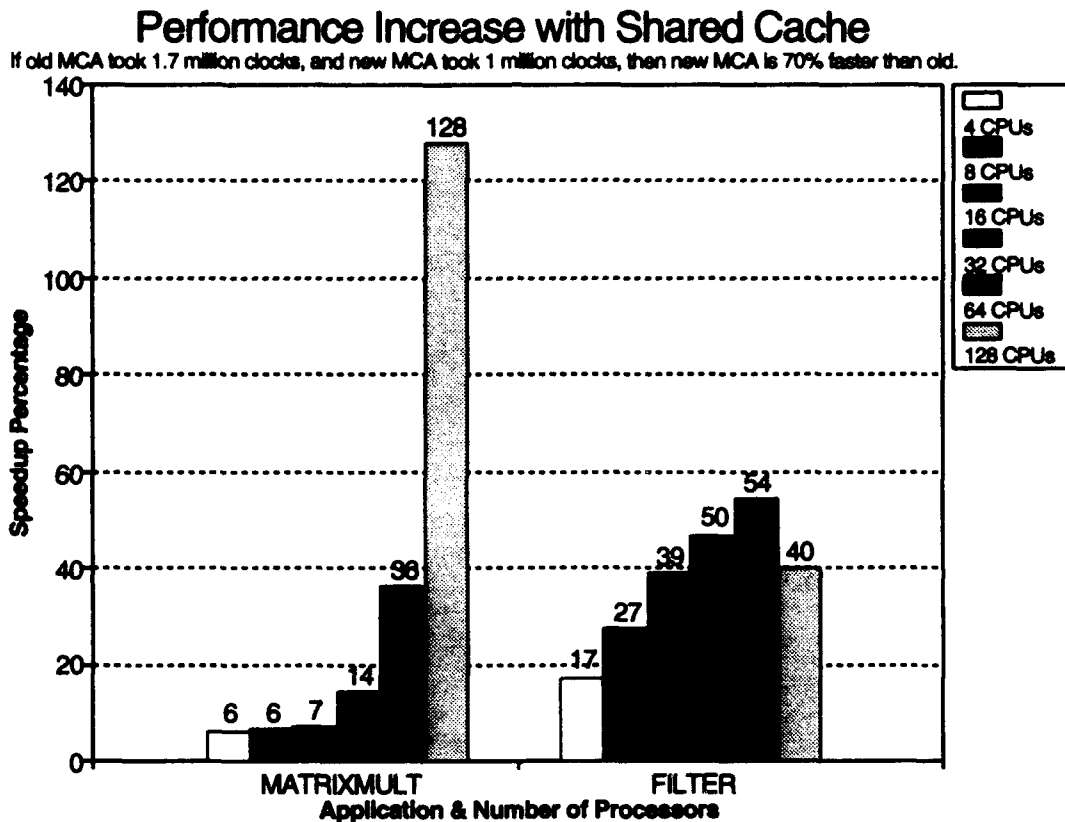
Performance Increase with Shared Cache

If old MCA took 1.7 million clocks, and new MCA took 1 million clocks, then new MCA is 70% faster than old.

Figure 6.10. **Performance Increase Attained by Caching Shared Variables.** The formula used for speedup is (Exold / Exnew) = 1 + (s / 100), where s is the speedup, and Exold, Exnew are the execution times for the baseline simulator and shared cache simulations, respectively.

6.17

Program Name	Number of CPUs	Number of Instructions[1]	Baseline Percent Utilization	Percent Utilization w/ Shared Cache
MATRIXMULT	4	18.4	87.2	92.5
(128 x 128)	8	9.4	85.6	90.9
	16	4.9	82.2	88.0
	32	2.7	74.5	84.2
	64	1.5	56.2	75.4
	128	1.0	30.1	67.8
FILTER	4	2.21	81.4	95.1
(64 x 64)	8	1.21	70.6	89.7
	16	0.71	58.9	80.9
	32	0.46	49.7	72.9
	64	0.34	41.6	63.8
	128	0.28	39.3	54.7

Table 6.2: CPU Utilization for the Master Processor for the Six Test Configurations.

([1]in millions)

6.6. Hit Rates

Cache hit rates are an important concern because the hit rate is a major factor in overall performance. Figures 6.11 and 6.12 show the hit rate in the shared cache across all processors in the system. MATRIXMULT had an average hit rate of 78% across the six configurations; FILTER's average hit rate was 84%. Because invalidations occur, the shared hit rate is not expected to reach the high values attained in the private cache (private cache hit rates run over 99%).

As invalidations affect more processors, the hit rate is subject to decline. Consider: if processor A is using variable X, and processor B writes to the block containing X, then processor A will have a miss on the next reference to X, and fetch the block again. Such invalidations explain why the shared
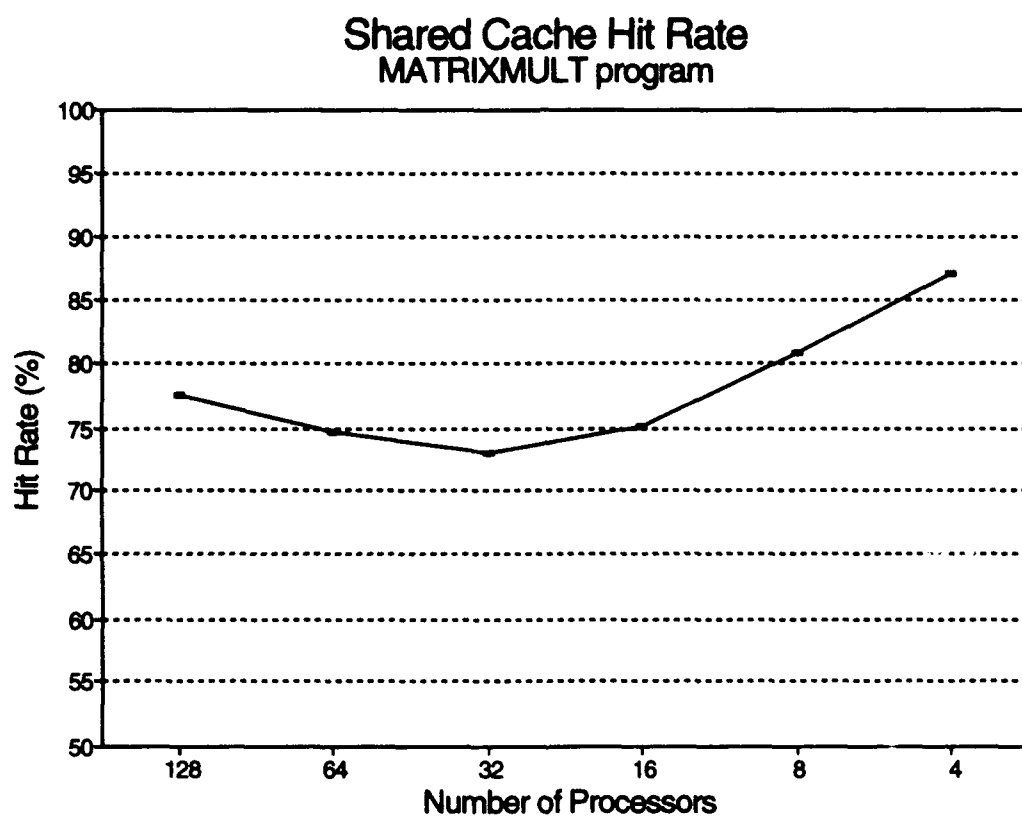
6.18

Shared Cache Hit Rate
MATRIXMULT program

Figure 6.11: Hit Rate in the Shared Cache, MATRIXMULT Program, with a
32K Cache, and Block Size of 32 Bytes.
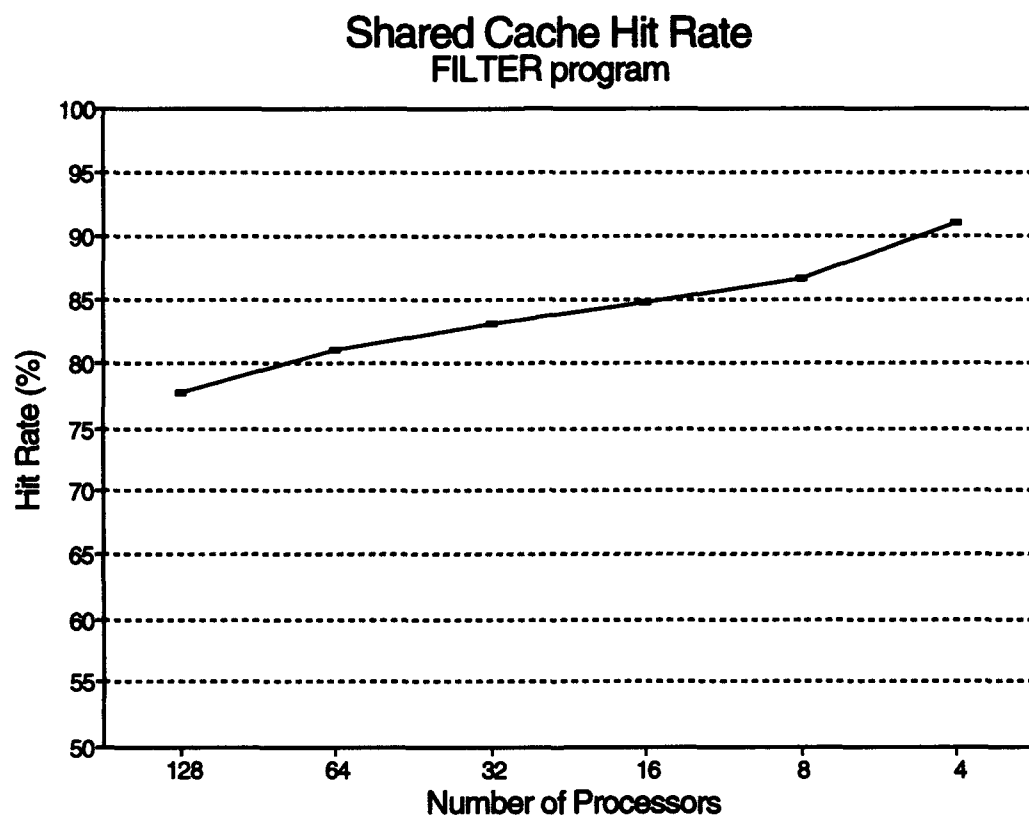
6.19

Shared Cache Hit Rate
FILTER program

Figure 6.12: Hit Rate in the Shared Cache, FILTER Program, with a 32K Cache, and Block Size of 32 Bytes.

cache hit rates do not approach 99%. Also, the hit rate tends to decline as more processors are added to a task, because there is greater possibility of an invalidation when cached blocks are more widely dispersed. While this trend occurred exactly for FILTER (Figure 6.12), MATRIXMULT (Figure 6.11) has an unexplained anomaly in that the hit rate bottoms out with 32 processors. What might cause this? Other factors affecting hit rate are the *overall size* of the cache (cache ejections increase when a cache is too small), and the number of EIBs during the task. While the *number of EIBs* remained roughly constant for FILTER (Figure 6.5), the total number of EIBs for MATRIXMULT is neither constant nor linear over the six test configurations, as Figure 6.4 revealed. A test performed for MATRIXMULT 8 demonstrated that a 256K size shared cache was needed to prevent unnecessary ejections (See Table 6.3), and only a 32K cache was used for the MATRIXMULT simulations.

| | FILTER 8 | | | MATRIX-MULT 8 | | |
Cache Size (K)	Private Hit Rate	Shared Hit Rate	Clocks (millions)	Private Hit Rate	Shared Hit Rate	Clocks (millions)
512	.99984	.86822	2.38	.999558	.93420	17.50
256	.99984	.86822	2.38	.999558	.93420	17.50
128	.99984	.86822	2.38	.999555	.93416	17.50
64	.99984	.86822	2.38	.998236	.93359	18.63
32	.99984	.86822	2.38	.986124	.80700	29.41
16	.99973	.85942	2.33	.985557	.80873	29.45
8	.99837	.78196	2.70	N/A	N/A	N/A

Table 6.3: Hit Rates for Various Shared and Private Cache Sizes, for Both FILTER 8 and MATRIXMULT 8. When the hit rate does not improve as the cache size is increased, then the smaller cache is sufficiently large. A 32K shared and private cache are sufficiently large for FILTER 8. MATRIXMULT 8 requires 256K caches before the hit rate no longer improves, although a 128K cache is almost sufficient; a sixth decimal place is needed to observe the difference in the private cache.

Therefore, the aforementioned anomaly is most likely caused by the non-linear EIB rate and the undersized cache, though the exact relationship between these factors and the results shown in Figure 6.12 is unknown. Finally, the fact that the shared cache had a higher hit rate for a 16K cache than for a 32K cache for MATRIXMULT 8 is a very surprising (and unexplained) result. This unusual behavior, unlikely to occur with private data, illustrates how invalidations and data partitioning are additional factors which impact the shared hit rate.

Section 4.2.2 mentioned that one potential benefit of establishing a separate cache for shared data is that the shared cache need not have the same hardware parameters as the private cache. This concept was tested and found to be true.

6.7. Varying Shared Cache Parameters

6.7.1. Devising the Experiment to Test Differing Cache Parameters

Line (block) size is one of the most important considerations for cache design. Too small a block size will result in too many unnecessary fetches. Because of the overhead required to send packets across the network, it is generally better to obtain more information per fetch. However, if a block size is too big, the cache becomes polluted, that is, data is put into the cache which is never used by the processor.

To test the potential benefit of differing cache parameters, FILTER 8 was used. As shown in Table 6.3, a 32K cache is sufficiently large for both the shared and private caches during FILTER 8. Having determined how large a cache is necessary, the next goal was to find the optimum hit rates as the line sizes varied in a sufficiently large cache. This information would then be

6.22

used to construct a custom design, to see if a performance gain could be realized by differing the shared and private cache line sizes.

6.7.2. Determining Optimum Line Sizes for FILTER 8

To ensure that the experiment would not be affected by too small a cache, both the private and the shared cache sizes were set to 64K (one increment bigger than the minimum sufficiently large cache). Using this size, the line sizes were varied between 8 and 128 (by powers of two). Table 6.4 shows the results of the five test runs. The peak hit rate for the shared cache occurred with a 64-byte block; for the private cache, a 128-byte block yielded the highest hit rate. The number of clocks required to complete the FILTER program is also listed for the three best runs.

LINE SIZE (bytes)	SHARED HIT RATE	PRIVATE HIT RATE	NUMBER OF CLOCKS TO COMPLETION
128	.856151	.999949	2,377,465
64	.883315	.999908	2,364,979
32	.868218	.999838	2,377,573
16	.832973	.999629	N/A
8	.774308	.999456	N/A

Table 6.4: Hit Rates and Completion Times for Varying Block Sizes in a 64K Cache, for FILTER 8.

6.7.3. Different Block Sizes for Each Cache

One more simulation was run. Each cache size was still 64K, but the block sizes differed; the private cache had a 128-byte block, while the shared cache used a 64-byte block. This simulation completed in less than 2.27 million clock cycles, over 95,000 cycles faster than the previous best time, thereby increasing the speedup over 3%. When this experiment was repeated

6.23

for the other FILTER configurations, the 128-byte private block and 64-byte shared block performed better than the matching block size simulations, for all FILTER configurations. A summary of the significant FILTER 8 runs is shown in Table 6.5. The results for all FILTER configurations are shown in Figure 6.13.

Although this experiment was only performed on one program, it proved that there is a potential performance benefit to be gained by differing the block sizes for shared and private data. The FILTER 8 finish times were nominally different when the cache line sizes were the same for both shared and private; a significant speedup caused by varying line sizes was only attained by customizing and varying the two line sizes. Such a benefit can only be realized by separating the two caches.

PRIVATE LINE SIZE	SHARED LINE SIZE	NUMBER OF CLOCKS TO COMPLETE	PERCENT SPEEDUP OVER BASELINE
32	N/A	3,030,922	(Baseline)
32	32	2,377,573	21.6%
64	64	2,364,979	22.0%
128	128	2,377,465	21.6%
128	64	2,269,141	25.1%

Table 6.5: Completion Times of FILTER 8 as Line Sizes Vary.

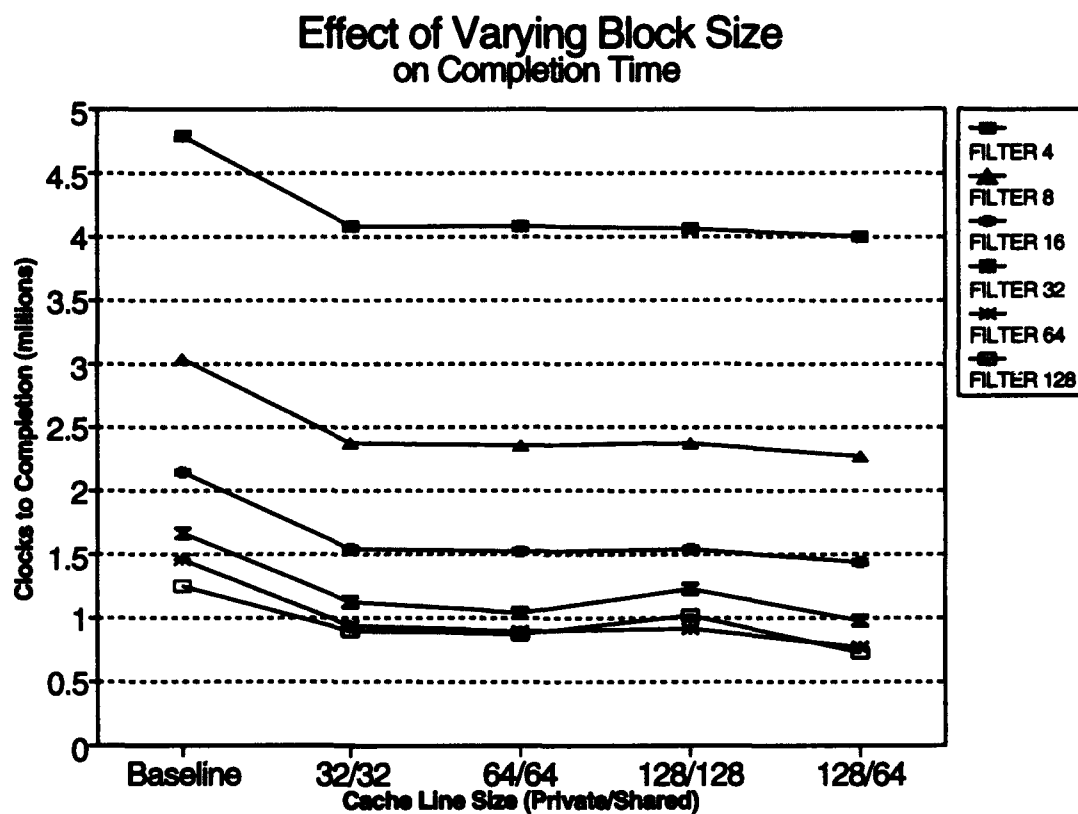**Effect of Varying Block Size on Completion Time**

Figure 6.13. Effect of Varying Block Sizes on FILTER Completion Time. In all cases, performance improved when the block sizes in the independent caches were not equal.

6.25

6.8. Conclusion

Initial testing has indicated that the original goal of the thesis, to improve the performance of the MCA through a shared coherent cache, has been accomplished. An overflow directory coherency protocol was originally proposed, but testing showed that the acknowledgement packets demanded by the directory hindered performance. Fortunately, the secondary notification scheme (extended invalidation broadcasts via a time-division multiplexed optical channel) performs very well by itself. Therefore, the proposed solution to the problem is the coherency protocol which uses no directory, but a TDM EIB channel.

The next chapter summarizes the lessons learned and suggests additional research.

CHAPTER 7

SUMMARY, CONCLUSIONS, AND FUTURE RESEARCH

7.1. Lessons Learned

This research contains results which are useful in the quest to find a scalable coherency protocol. First, it was shown that even a small directory is not a satisfactory solution if acknowledgement packets are required to be sent back to a centralized location. Additionally, it was shown that timedivision multiplexing is an effective way to collect acknowledgements. Experiment results showed that a TDM communication channel may be a very effective ingredient for solving the well-known cache coherency problem. Additional research which further tests the feasibility of this approach would be very interesting.

Second, the idea of separating shared and private caches was proposed. As massively parallel systems become more widespread, this could be an effective design decision regardless of the protocol chosen. It allows a more customized shared cache, and frees the private cache of coherency traffic. Some preliminary results showed that there are benefits to this separation. Among them, it was shown customized line sizes between the two separate caches can improve performance.

Third, the research of Gupta and Weber [Gup92] was tested and independently confirmed. Although this does not provide any new information, it does solidify their findings.

Fourth, some of the problems associated with maintaining coherency across a network were examined. Namely, there is no guarantee that important information will arrive at a destination node in a timely manner.

7.1

One problem encountered was stale data arriving at a cache after an invalidation has occurred. This is a possibility when a requested packet is delayed at the transmitter. The reservation required scheme presented in this thesis is an innovative way to overcome this problem.

7.2. Future MCA Research

There are four programs in the MCA test suite, but only two were used to evaluate the shared cache performance. This is because the simulator still has some errors introduced during the software modifications. This research progressed simultaneously with another project which allowed the MCA simulator to simulate running multiple tasks at the same time. Thus, the changes were very extensive, and a few errors remain. Once the remaining errors are fixed, it would be prudent to run tests on the remaining two applications, to ensure that the other two programs do not show dramatically different results which might nullify some of the conclusions already drawn from this research.

The MCA coherency protocol evolved from a directory-based scheme to a snooping-based scheme, since the EIBs are broadcast over a common bus. It would be interesting to enhance the protocol so that it accounts for exclusiveness. Instead of maintaining a channel-based directory, as was originally proposed, the shared memory nodes could maintain an overflow directory based on processor. The directory would be of length one. Essentially, if one and only one processor has cached a shared variable, then the EIB could be skipped, and the write-grant issued immediately. This might improve the protocol.

7.3. Other Topics of Interest

Acknowledgement packets cripple a centralized directory and hurt its scalability. However, there may be a way to build some sort of hierarchical scheme where the acknowledgement packets do not overwhelm a particular node.

A comprehensive study trying to determine ideal shared block sizes would also be interesting. Smith [Smi85] has already performed such an analysis, but that experiment did not study shared and private data independently. The potential advantage of dissimilar block sizes has already been demonstrated. A study which is more focused on shared data behavior could be used determine ideal shared cache parameters.

Compile-time analysis could be used to make a hardware protocol more efficient. Min and Baer [Min92] have stated that a hybrid between a hardware and software solution would be an interesting research topic. They propose letting a hardware protocol help a software protocol. The problem could also be studied from the opposite perspective, letting the software analysis assist the hardware protocol. Write-runs and memory-reads are two items in particular which could be flagged at compile time and improve the performance of a hardware protocol. Exploiting write-runs would reduce the amount of network traffic and eliminate some meaningless invalidations. Memory reads could possibly prevent unfruitful ejections, thereby increasing the shared cache hit rate.

7.4. Conclusion

This thesis proposed an overflow directory protocol for the MCA. The overflow directory protocol's secondary notification scheme used a TDM

optical channel for coherency traffic. Shared data was cached in an independent cache.

The overflow directory did not work well, because the generated acknowledgement packets caused too much bus contention. However, preliminary results indicate that two other ideas have merit, and deserve further study. In particular, TDM communication seems an efficient way to broadcast cache invalidations and rapidly collect acknowledgements, without burdening the overall interconnection network. A TDM scheme could be developed for any massively parallel machine, electrical or optical. In fact, time-division multiplexing might be an effective instrument in developing a scalable coherency protocol. Furthermore, the advantages of split caches (for shared and private data) have already been discussed. This design decision could be incorporated into any parallel architecture to realize these benefits.

BIBLIOGRAPHY

[Arc86] J. Archibald and J.-L. Baer. "Cache Coherence Protocols: Evaluation Using a Multiprocessor Simulation Model." In *ACM Trans. Comp. Sys.*, vol. 4, no. 4, pp. 273-298, Nov. 1986.

[Cen78] L.M. Censier and P. Feautrier. "A New Solution to Coherence Problems in Multicache Systems." In *IEEE Trans. on Computers*, pp. 1112-1118, 1978.

[Cha90] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. "Directory-Based Cache Coherence in Large-Scale Multiprocessors." In IEEE *Computer*, vol. 23, no. 6, pp. 49-58, Jun. 1990.

[Che88] H. Cheong and A.V. Veidenbaum. "A Cache Coherence Scheme with Fast Selective Invalidation." In *Proc. of the 15th Int. Sym. on Comp. Arch. (IEEE)*, pp. 299-307, 1988.

[Egg88] S.J. Eggers and R.H. Katz. "A Characterization of Sharing in Parallel Programs and Its Application to Coherency Protocol Evaluation." In *Proc. of the 15th Int. Sym. on Comp. Arch. (IEEE)*, pp. 373-382, 1988.

[For93] F. Forghieri, A. Bononi, and P.R. Prucnal. "Novel Packet Architecture for Ultrafast All-Optical Networks." In *OFC/IOOC '93 Technical Digest*, Feb 21-26, 1993, vol. 4, San Jose, CA. Pp. 198-199, 1993.

[Fra84] S.J. Frank. "Tightly Coupled Multiprocessor System Speeds Memory Access Times" In *Electronics*, vol. 54, no. 1, pp. 164-169, Jan. 12, 1984.

[Gla93] B.S. Glance, J.M. Wiesenfeld, U. Koren, and R.W. Wilson. "New Advances on Optical Components Needed for FDM Optical Networks." In *IEEE Journal of Lightwave Tech.*, vol. 11, no. 5/6, pp. 882-889, May/Jun. 1993.

[Goo83] J.R. Goodman. "Using Cache Memory to Reduce Processor-Memory Traffic." In *Proc. of the 10th Int. Sym. on Comp. Arch. (IEEE)*, pp. 124-131, 1983.

[Gup92] A. Gupta and W.-D. Weber. "Cache Invalidation Patterns in Shared-Memory Multiprocessors." In *IEEE Trans. on Computers*, vol. 41, no. 7, pp. 794-810, Jul. 1992.

[Han88] G. Hanke. "Signal Processing at 4.5 GBit/S with SI-ICs for Optical Transmission Systems." In *MTT-S Digest (IEEE)*, pp. 853-855, 1988.

[Hen90] John L. Hennessy and David A. Patterson. Computer Architecture: A Quantitative Approach. Morgan Kaufmann Publishers, San Mateo, CA, 1990.

[Hil89] M.D. Hill and A.J. Smith. "Evaluating Associativity in CPU Caches." In *IEEE Trans. on Computers*, vol. 38, no. 12, pp. 1612-1630, Jan. 1990.

[Kat85] R.H. Katz, S.J. Eggers, D.A. Wood, C.L. Perkins, and R.G. Sheldon. "Implementing a Cache Consistency Protocol." In *Proc. of the 12th Int. Sym. on Comp. Arch. (IEEE)*, pp. 276-283, 1985.

[Kar93] M.J. Karol and B. Glance. "Performance of the PAC Optical Packet Network." In *IEEE Journal of Lightwave Tech.*, vol. 11, no. 8, pp. 1394-1399, Aug. 1993.

[KSR92] Kendall Square Research. "Technical Summary of the KSR1." Kendall Square Research, Waltham, MA, 1992.

[Len92] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, et. al. "The Stanford Dash Multiprocessor." In IEEE *Computer*, vol. 25, no. 3, pp. 63-79, Mar. 1992.

[OKr90] B.W. O'Krafka and A.R. Newton. "An Empirical Evaluation of Two Memory-Efficient Directory Methods." In *Proc. of the 17th Int. Sym. on Comp. Arch. (IEEE)*, pp. 138-147, 1990.

[Min92] S.L. Min and J.-L. Baer. "Design and Analysis of a Scalable Cache Coherence Scheme Based on Clocks and Timestamps." In *IEEE Trans. on Parallel and Dist. Sys.*, vol. 3, no. 1, pp. 25-43, Jan. 1992.

[Pap84] M.S. Papamarcos and J.H. Patel. "A Low-Overhead Coherence Solution for Multiprocessors with Private Cache Memories." In *Proc. of the 11th Int. Sym. on Comp. Arch. (IEEE)*, pp. 348-354, 1984.

[Prz88] S. Przybylski, M. Horowitz, and J. Hennessy. "Performance Tradeoffs in Cache Design." In *Proc. of the 15th Int. Sym. on Comp. Arch. (IEEE)*, pp. 220-298, 1988.

[Smi82] A.J. Smith. "Cache Memories." In *Computing Surveys (ACM)*, vol. 14, no. 3, pp. 473-530, September 1982.

[Smi85] J.E. Smith and J.R. Goodman. "Instruction Cache Replacement Policies and Organizations." In *IEEE Trans. on Computers*, vol. C-34, no. 3, pp. 234-241, March 1985.

[Smi87] A.J. Smith. "Line (Block) Size Choice for CPU Cache Memories." In *IEEE Trans. on Computers*, pp. 1063-1075, vol. C-36, no. 9, Sept. 1987.

[Ste90] P. Stenström. "A Survey of Cache Coherence Schemes for Multiprocessors." In IEEE *Computer*, vol. 23, no. 6, pp. 12-24. 1990.

[Swe86] P. Sweazey and A.J. Smith. "A Class of Compatable Cache Consistency Protocols and Their Support by the IEEE Futurebus." In *Proc. of the 13th Int. Sym. on Comp. Arch. (IEEE)*, pp. 414-423, 1986.

[Tha88] C.P. Thacker, L.C. Stewart, and E.H. Satterthwaite, Jr. "Firefly: A Multiprocessor Workstation." In *IEEE Trans. on Computers*, vol. 37, no. 8, pp. 909-920, Aug. 1988.

[Tuc87] R.S. Tucker, G. Eisenstein, S.K. Korotky, et. al. "Optical Time-Division Multiplexed Fiber Transmission Using Ti:LiNbO$_3$ Waveguide Switch/Modulators." In *Procedings, Int. Sym. on Global Comm.*, pp. 1302-1304, 1987.

[Wai91] T.S. Wailes and D.G. Meyer. "Multiple Channel Architecture: A New Optical Interconnection Strategy for Massively Parallel Computers." In *IEEE Journal of Lightwave Tech.*, vol. 9, no. 12, pp. 1702-1716, Dec. 1991.

[Wai92] T.S. Wailes. "Multiple Channel Architecture: A New Optical Interconnection Strategy for Massively Parallel Computers." Ph.D. thesis, Purdue University, 1992.

VITA

Captain John A. Reisner was born 12 August 1963 on Cape Cod, Massachusetts. He graduated from Chatham High School in 1981. He earned a Bachelor of Science degree in Computer Science from the University of Lowell in 1985, where he was commissioned upon graduation as an Air Force Second Lieutenant. From 1986 until 1992 he served as a Computer Analyst, supporting the software platform used by the battle staff on the Strategic Air Command's Airborne Command Post, also known as *Looking Glass*. During part of that assignment, he was put on flight crew as an Airborne Computer Analyst. In May of 1992 he entered the Graduate School of Engineering, Air Force Institute of Technology, to earn a Master of Science degree in Computer Systems.

Permanent Address:

21 Carolyn Drive
Chatham, MA 02633

REPORT DOCUMENTATION PAGE

1. AGENCY USE ONLY (Leave blank)	2. REPORT DATE	3. REPORT TYPE AND DATES COVERED
	December 1993	Master's Thesis

4. TITLE AND SUBTITLE

DESIGN OF A SHARED COHERENT CACHE
FOR A MULTIPLE CHANNEL ARCHITECTURE

5. FUNDING NUMBERS

6. AUTHOR(S)

John A. Reisner, Captain, USAF

7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES)

Air Force Institute of Technology
WPAFB, OH 45433

8. PERFORMING ORGANIZATION REPORT NUMBER

AFIT/GCS/ENG/93D-19

9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES)

10. SPONSORING/MONITORING AGENCY REPORT NUMBER

11. SUPPLEMENTARY NOTES

12a. DISTRIBUTION/AVAILABILITY STATEMENT

Distribution Unlimited

12b. DISTRIBUTION CODE

13. ABSTRACT (Maximum 200 words)

Abstract

The Multiple Channel Architecture (MCA) is a recently proposed computer architecture which uses fiber optic communications to overcome many of the problems associated with interconnection networks. There exists a detailed MCA simulator which faithfully simulates an MCA system, however, the original version of the simulator did not cache shared data. In order to improve the performance of the MCA, a cache coherency protocol was developed and implemented in the simulator. The protocol has two features which are significant: (1) a time-division multiplexed (TDM) communication bus is used for coherency traffic, and (2) the shared data is cached in an independent cache. The modified simulator was then used to test the protocol. Two applications and six test configurations were used throughout the testing. Experiment results showed that the protocol consistently improved system performance. Also, a proof-of-concept experiment indicated that performance improvements can be attained by varying cache parameters between the independent shared and private data caches.

14. SUBJECT TERMS

cache design, cache coherence, shared memory
multiprocessors, time-division multiplexing

15. NUMBER OF PAGES

111

16. PRICE CODE

17. SECURITY CLASSIFICATION OF REPORT	18. SECURITY CLASSIFICATION OF THIS PAGE	19. SECURITY CLASSIFICATION OF ABSTRACT	20. LIMITATION OF ABSTRACT
Unclassified	Unclassified	Unclassified	UL