AN EMPIRICAL STUDY OF THREE HARDWARE CACHE CONSISTENCY SCHEMES FOR LARGE SHARED MEMORY MULTIPROCESSORS

by

Brian W. O'Krafka

Memorandum No. UCB/ERL M89/62

22 May 1989

COVER PAGE

AN EMPIRICAL STUDY OF THREE HARDWARE CACHE CONSISTENCY SCHEMES FOR LARGE SHARED MEMORY MULTIPROCESSORS

by

Brian W. O'Krafka

Memorandum No. UCB/ERL M89/62

22 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

AN EMPIRICAL STUDY OF THREE HARDWARE CACHE CONSISTENCY SCHEMES FOR LARGE SHARED MEMORY MULTIPROCESSORS

by

Brian W. O'Krafka

Memorandum No. UCB/ERL M89/62

22 May 1989

ELECTRONICS RESEARCH LABORATORY

College of Engineering
University of California, Berkeley
94720

Abstract

In this report an empirical study of three hardware schemes for managing caches and main memory in large shared memory multiprocessors is presented. The first scheme is a simple one in which shared writeable data is not cached at all. The second scheme is a version of Censier and Feautrier's directory method in which cache blocks are sectored into sub-blocks to reduce tag overhead. The third scheme is a modification of the second in which main memory is distributed among the processors to act as a second level cache; the motivation of this scheme is to increase the effective cache size to reduce average memory access time and network traffic. The larger effective cache size of the third scheme is provided at the expense of considerable added complexity in the coherence protocol. The three schemes are compared on the basis of simulation results obtained from complete architectural simulations of three different multiprocessors executing four benchmark programs. In all simulations a multiprocessor with 64 processors was used. Cache sizes of 100 kilobytes for schemes one and two, and 1 megabyte for scheme three were assumed. Simulation results indicate that for the benchmarks considered the second coherence scheme reduces both average memory access time and network traffic by over a factor of two, relative to the first scheme. Although for three of the benchmarks the third coherence scheme performs almost as well as the second, it performs very poorly for the fourth benchmark because of a contention problem in which a large number of blocks accumulate in a single memory bank. Results for different block sizes and constant sub-block size show that increasing the block size increases the number of misses for the second and third schemes. The increase in miss ratio, however, does not significantly degrade average memory access time or network traffic. This is because the fraction of network traffic caused by references to synchronization variables (which always bypass the cache) is comparable to the fraction caused by cache misses. The average number of invalidations issued per shared write was found to be very high for the sectored schemes, approaching half the number of processors. For sectoring schemes to be more effective it appears that caches may have to prematurely displace blocks that draw excessive invalidation traffic. In addition to providing comparative empirical data for these coherence techniques, the results also demonstrate the viability of complete architectural simulation as a means for evaluating multiprocessor designs.

Acknowledgements

I would like to acknowledge my research advisor, Professor Richard Newton, for providing the idea that started this research, for giving me the opportunity to pursue it, and for being a source of encouragement on many occasions.

I would also like to thank Professor Alan Smith for the many useful criticisms that came out of his thorough review of this report. Gregg Whitcomb and Abhijit Ghosh served as proofreaders, for which I am grateful.

I thankfully acknowledge the funding provided for this project by the following organizations: the Natural Sciences and Engineering Research Council of Canada, Digital Equipment Corporation, Hewlett Packard, and the Defense Advanced Research Projects Agency (under contract N00039-C-87-0182).

My wife, Audrey, shared my experiences during times of frustration as well as times of achievement, and for her longsuffering support I am deeply grateful.

Contents

List of Figures

List of Tables

Chapter 1

INTRODUCTION

Shared-bus, shared-memory multiprocessors are gaining commercial acceptance by offering high performance at reduced cost for applications with exploitable parallelism. By efficiently supporting the shared memory programming paradigm, these machines are usually much easier to program than those supporting message passing because the programmer does not have to worry about distributing data across multiple memories; the shared memory paradigm is thus especially attractive for applications with memory reference characteristics that are not easily predicted. The shared memory *programming paradigm* should be distinguished from the *hardware* on which the paradigm is implemented, which may or may not support shared memory directly. For an implementation of the shared memory programming paradigm to be effective the average memory access time should be similar to the speed of the uniprocessors used to construct the multiprocessor. Any hardware that is intended to provide effective support of the shared memory programming paradigm must therefore provide low latency memory accesses in addition to high bandwidth. To achieve this, many shared memory multiprocessor desigus have used hardware specifically designed for the shared memory programming paradigm [P*85,G*83b,Seq84,Ros85,G*83a,SS86].

Many commercial shared memory offerings, including those by Sequent [Seq84] and Encore [Ros85], use efficient caching techniques known as *snooping cache protocols* [AB86] to provide effective shared memory systems at the hardware level. Unfortunately, these techniques rely on the use of a shared bus as the processor interconnection network, so the number of processors is restricted to the maximum number that can be supported by a single bus, probably 50 or less [A*88]. While much research has been reported investigating various snooping cache protocols [TS87,PP84,Y*85,EK88] , comparatively little has been done to develop alternative methods to support shared memory in much larger machines with 50 - 1000 processors.

Since implementations of shared memory in hardware are usually constructed from a distributed collection of single ported memories, memory controllers and network elements, they can only approximate the ideal, which can be described as:

1. offering simultaneous access by all processors for any combination of reads and writes;
2. having negligible access delay (not including any queuing delay attributed to the failure to meet 1);

The ideal shared memory should also not require a programmer to manually distribute data to exploit characteristics of the underlying implementation, such as the presence of fast buffers among slow bulk memories; the mapping of virtual addresses to their physical locations within possibly many constituent physical memories should be transparent to the user. Clearly, any implementation attempting to satisfy these requirements directly (with, for example, a large, fast multi-ported ECL memory) would be prohibitively expensive.

The degree to which the first requirement can be met is determined by the choice of interconnection network and the number of memory modules. Multiple, simultaneous memory accesses are typically restricted in two ways. First, networks usually only support a subset of the total set of access permutations. Second, the memory modules usually limit the number of simultaneous accesses. The net effect of both restrictions is increased delay. The latter limitation may be reduced by the use of a combining network [G*83b] in which simultaneous accesses to the same address can be "combined" in the network before reaching the memory module. Although a study [PN85] suggests that combining networks are very effective, it also points out that they are very expensive. Both limitations may be relieved in some cases by using a memory hierarchy in which cache memories are provided at each processor [Smi82]. Since caches permit memory references to be satisfied without using the network, the first restriction can be overcome. If multiple copies of data are permitted to exist in a number of caches simultaneously, the second restriction can be overcome.

The effective delay of a shared memory implementation, neglecting queuing effects, can also be reduced using a memory hierarchy. With fast caches associated with each processor and well-behaved memory reference activity, most references can be satisfied locally and avoid network and main memory delay.

Cache memories thus appear to be a cost effective way to implement a shared memory that approximates the ideal. When caches are used in a multiprocessor, however, some method must be used to ensure that the result of a write is reflected in all of the caches holding copies of the affected data. This is known as the cache consistency problem [Smi82]. Cache consistency methods proposed to date may be divided into four classes: those in which shared writeable data is uncached, snooping protocols, directory schemes and software assisted techniques [AB84].

The first class requires the least amount of hardware support but offers the poorest performance for applications in which a large fraction of memory references are to shared data.

The class of snooping protocols includes the well established assortment of shared-bus coherence schemes [K*85,TS87,Goo83,RS84,McC84,SS86] and several extended schemes which support more processors using collections of busses connected hierarchically [Wil87] or in a cube [GW88]. The key distinction of this class of consistency schemes is that any processor action which may affect other cached copies of data must be broadcast to all caches. For example, if a processor writes to shared writeable data, all other cached copies must be either invalidated or updated. The various schemes differ in the amount of state maintained for each cache block and in whether copies of data are invalidated or updated on a write. The amount of state, in turn, affects the efficiency of the protocol.

The class of directory schemes is unique in that a directory containing block state is associated with the main memory [AB84,Smi82]. All processor references which may affect other caches must go through the main memory, using the directory to determine which caches, if any, are affected. Since directory methods use broadcasting less frequently, they are potentially useful in much larger multiprocessors. There is also no fundamental dependence on a particular type of interconnection network as there is in snooping protocols.

Most software solutions to the cache coherence problem depend upon sophisticated compilers to insert cache invalidation instructions [CV88]. The IBM RP3 [P*85], University of Illinois Cedar [G*83a], and New York University Ultracomputer [G*83b] projects all use software solutions to the cache coherence problem. A typical software coherence strategy involves the tagging of shared writeable data and the insertion of cache invalidation instructions at the end of parallel computation units, such as parallel DO-loops. At the end of such a unit each cache invalidates all copies of the shared data affected by the unit, writing back all dirty blocks to main memory. The schemes vary in the way in which computation units are specified and in the hardware support used for bulk invalidations of shared data. A considerably different software-assisted scheme has been suggested by Smith [Smi85] in which special translation lookaside buffer entries are augmented with "one-time identifiers". Although software methods have no run-time communication overhead, they offer non-optimal performance because of an increased number of cache misses due to unnecessary invalidations, and added processor delay while the bulk invalidations take place. Relatively little performance data has been published concerning software methods, so their effectiveness remains unclear.

In this report the effectiveness of three hardware coherency schemes is evaluated:

1. A scheme in which shared writeable data is not cached.

2. A variation of the Censier and Feautrier directory scheme [CF78].

3. A novel extension of the Censier and Feautrier scheme which permits dynamic assignment of blocks among a distributed main memory.

These schemes are evaluated using simulation results based on a set of benchmark programs which were designed from the outset for implementation on shared memory multiprocessors. Two of the four benchmarks are design aids for integrated circuits, chosen because they represent a class of applications which are generally difficult to program without the shared memory paradigm, are computationally intensive, are of widespread interest, and have considerable exploitable concurrency. The other two benchmarks are a simple multiprocessor simulator and a very simple test program.

The report is organized as follows. Chapter 2 presents the specifications of the three hardware-based consistency schemes, which are evaluated using the methodology described in Chapter 3. Chapter 4 summarizes the simulation results, from which conclusions are drawn in Chapter 5.

Chapter 2

Three Approaches to Hardware-based Cache Consistency

In this chapter three hardware-based approaches to enforcing cache consistency in large shared memory multiprocessors are presented. For each of the schemes it is assumed that the requests and responses of Table 2.1 are issued and accepted by a processor ("CPU"). A SYNCOP refers to any operation that operates on synchronization variables, where a synchronization variable is any variable used to control the interaction of multiple processes. Examples of SYNCOP's include test-and-set [AS83] and fetch-and-add [G*83b]. These primitive operations can be used to construct higher level synchronization facilities such as spin-locks, semaphores or communicating processes [AS83]. It is further assumed that all interprocess synchronization is performed using explicitly-declared synchronization variables, operated upon using only SYNCOP operations. This is necessary so that synchronization variables are not cached.

2.1 No Caching of Shared Writeable Data

For this scheme it is assumed the multiprocessor architecture of Figure 2.1, in which CPU requests are sent to memory controllers that access a cache or main memory, via an interconnection network, as appropriate. In this first coherence technique, shared

Transaction	Description
READ(addr)	Read data at *addr*
WRITE(addr, data)	Write *data* to *addr*
SYNCOP(addr, data)	Operate on the synchronization variable at *addr* using *data*
DATA(data)	Data returned from a READ or SYNCOP
ACK()	Acknowledgement that a WRITE has completed

Table 2.1: CPU Requests

writeable data is tagged uncacheable. The memory controllers must bypass their caches and access main memory for all references to non-cacheable data. In this simple scheme the main memory must be able to handle cache requests for block contents and cache requests to displace blocks. Table 2.2 shows the network transactions that can take place between a memory controller and a main memory; some of the transactions are used only in the other coherence protocols and are explained in the appropriate sections. Only the DISPLACE and MREAD transactions are required for this scheme. Since up to 40 % of all references can be to shared writeable data, this method offers the poorest performance of the three schemes, but provides a useful reference point for evaluating the others.

2.2 Modified Censier and Feautrier Protocol

This version of the Censier and Feautrier directory method [CF78] incorporates a modification to reduce the large amount of extra memory needed to store the large tags required by the original protocol. Censier and Feautrier's scheme as originally presented in [CF78] is presented first, followed by a discussion about the modification. In both descriptions the multiprocessor architecture of Figure 2.1 is assumed.

2.2.1 Basic Censier and Feautrier Scheme

In this scheme physical memory is divided into blocks of fixed size. Each block of main memory is associated with a directory entry (or tag) containing 1 bit per cache, a single bit indicating whether or not the block is modified, and a lock bit (Figure 2.2). A block is always in one of these three states:

1. ABSENT: no cache holds a copy (all cache bits in the directory entry are 0, and the modified bit is 0; lock bit is 0);

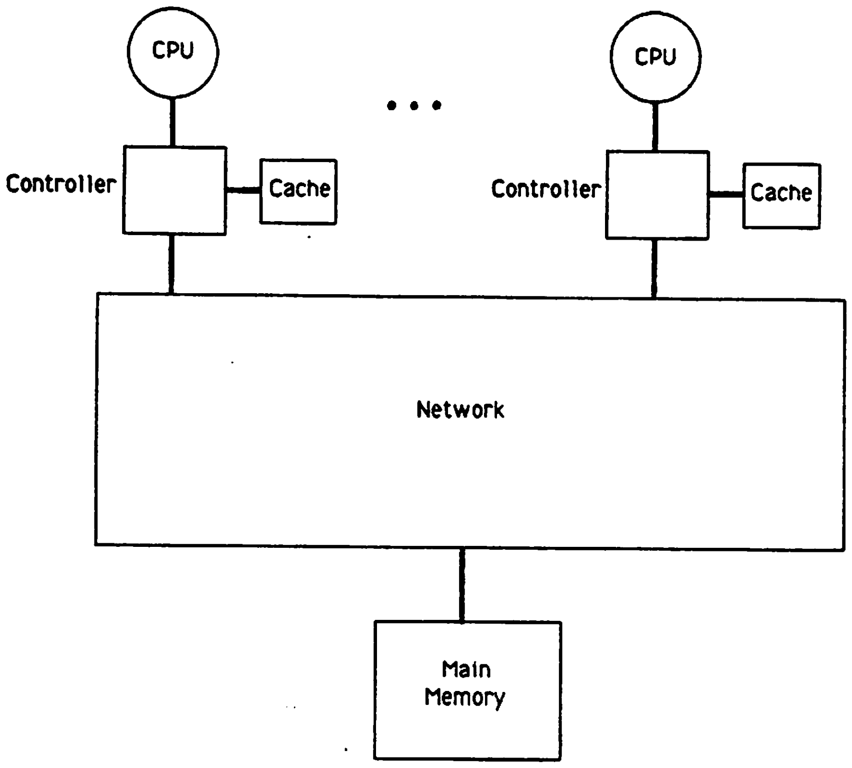

Figure 2.1: Multiprocessor Architecture for NOCACHE and CANDF Schemes

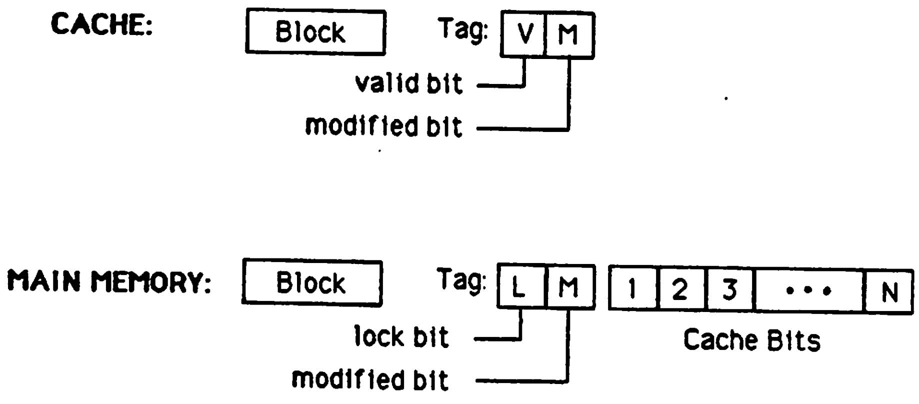

Figure 2.2: Tags for Basic Censier and Feautrier Protocol

2. PRESENT: one or more caches hold copies, and the block is unmodified (one or more cache bits in the directory entry are 1, and the modified bit is 0; lock bit is 0);

3. PRESENTM: exactly one cache has a copy and it is modified (exactly one cache bit is 1 and modified bit is 1; lock bit is 0);

4. LOCKED: an operation on this block is currently in progress (lock bit is 1);

In like manner, each cache block is associated with a cache directory entry consisting of a valid bit and a modified bit (Figure 2.2). Cache blocks may be in one of these states:

1. INVALID: the contents of the cache block are invalid (valid bit is 0);

2. VALID: the contents of the cache block are valid and unmodified (valid bit is 1 and modified bit is 0);

3. VALIDM: the contents of the cache block are valid and modified (valid bit is 1 and modified bit is 1). This state implies that this cache has the only valid copy of the block in the entire multiprocessor.

A cache consistency protocol is defined by the set of actions taken by the memory controllers and the main memory for each different combination of processor request, cache block state, and main memory state. Figures 2.3 to 2.11 presents the details of the Censier and Feautrier protocol in flowchart form.

If a processor issues a read and the local cache block of the data is valid, no main memory access is needed and the data is simply read from the cache. If a block for the referenced data does not exist ("miss" state in Figure 2.3) a block must be assigned and its old data displaced to main memory. The missed reference is then handled as if the block was invalid: a read transaction is issued to the main memory. If the main memory block is in an unmodified state, the block contents are returned to the requesting memory controller. If the main memory block is modified, the block contents are read from the single "owning" cache, written to main memory, and forwarded to the requestor. In all of these cases the cache and main memory entries have their states updated as shown in Figures 2.3 and 2.4.

When a processor issues a write, it can only be satisfied locally if the local cache block is VALIDM. If the local cache state is VALID, an invalidate transaction is sent to the

Transaction	Description
MREAD(addr)	Read a block from main memory
CREAD(addr)	Read a block from a cache
MINVAL(addr)	Invalidate a sub-block at the main memory
MINVAL-FETCH(addr)	Invalidate a sub-block at the main memory and fetch its contents
CINVAL(addr)	Invalidate a sub-block at a cache
CINVAL-FETCH(addr)	Invalidate a sub-block at a cache and fetch its contents
MDATA(data)	Sub-block data returned from main memory
CDATA(addr, data)	Sub-block data returned from a cache
CACK(addr)	Acknowledgement returned by a cache in response to CINVAL
MACK()	Acknowledgement returned by main memory to MINVAL
MSYNCOP(addr, data)	Operate on a synchronization variable at the main memory
MSYNCDATA(data)	Data returned from main memory in response to MSYNCOP
DISPLACE(addr)	Displace a block to main memory
FAIL(addr)	Indicates that a transaction is already in progress for the referenced block; the requestor must try again

Table 2.2: Network Transactions

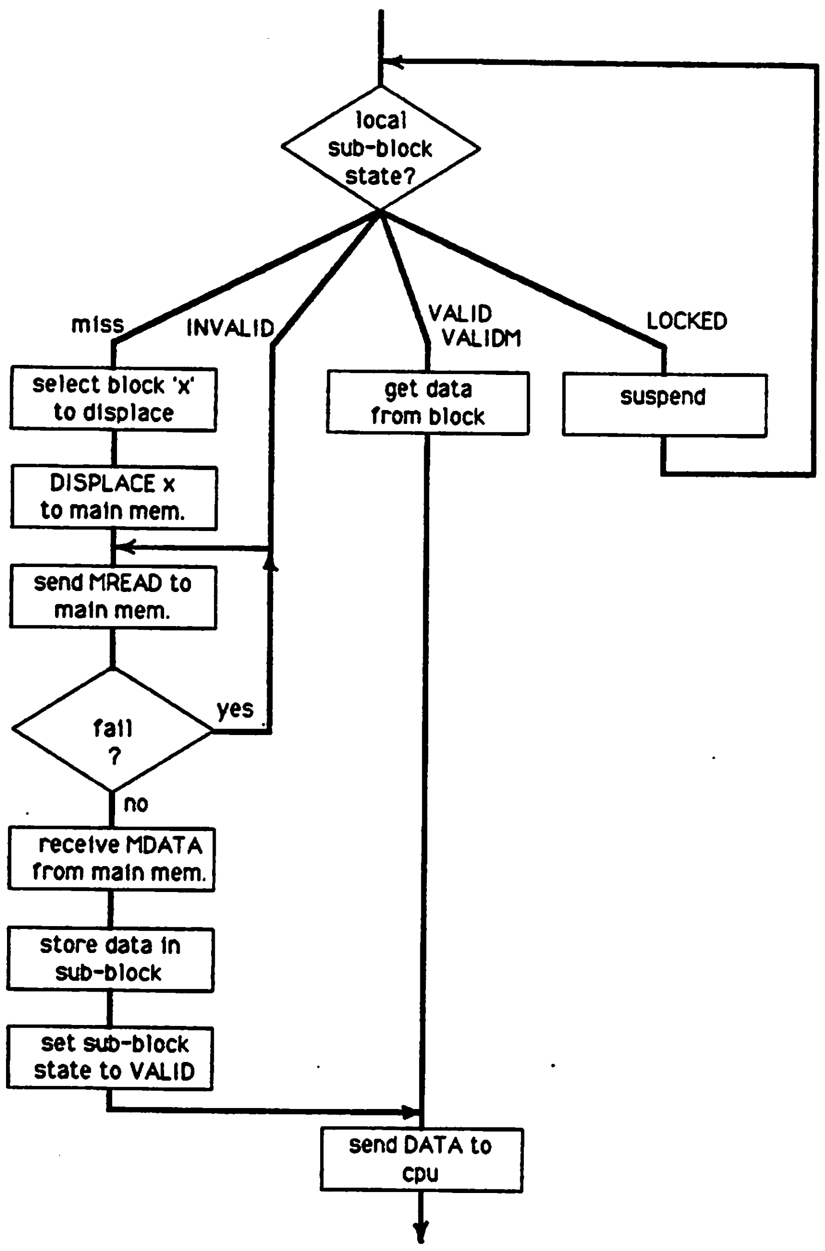

Figure 2.3: Cache Read

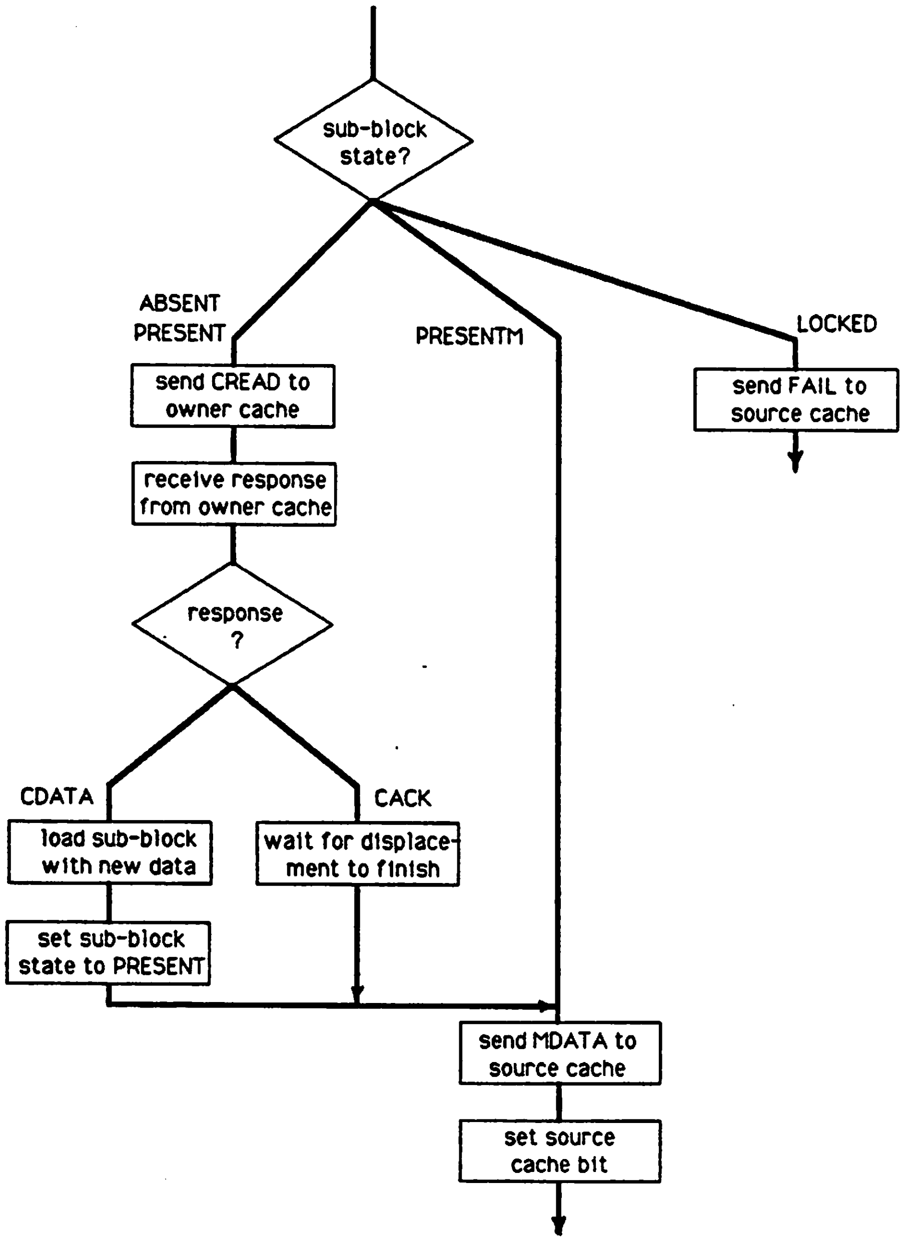
sub-block state?
ABSENT
PRESENT
PRESENTM
LOCKED
send CREAD to owner cache
receive response from owner cache
response ?
CDATA
CACK
load sub-block with new data
set sub-block state to PRESENT
wait for displacement to finish
send FAIL to source cache
send MDATA to source cache
set source cache bit

Figure 2.4: Memory Read

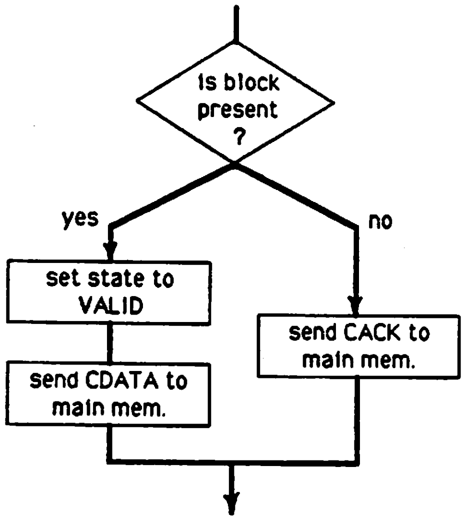

Figure 2.5: Cache Read from Memory

main memory which, if other caches have copies (ie: main memory state is PRESENT or PRESENTM), issues invalidations to them. If, however, the local cache misses or the block is invalid, the controller issues an invalidate-fetch transaction to the main memory. A fetch is required here so that the portion of the block untouched by the write is made valid. The main memory sends invalidations to caches with copies and if the block is modified, fetches the current data, updates itself, and forwards the data to the requestor. As before, the states of cache and main memory blocks are updated as shown in Figures 2.6, 2.7 and 2.8.

It was assumed that references to synchronization variables bypass the cache completely and are always handled at the main memory.

Block displacements are always sent to the main memory which updates its directory as appropriate. If a cache displaces a VALIDM block, the block contents must be written back.

The Censier and Feautrier scheme is well-suited to large multiprocessors because it does not depend on the use of broadcasts, and hence is not designed for a particular network, and permits the main memory and its directory to be interleaved. Although the communication overhead could be excessive if many blocks reside in many caches, the scheme's greatest drawback is the severe memory overhead introduced by the large number of cache bits in the main memory tags. As an example, a system with 100 processors requires

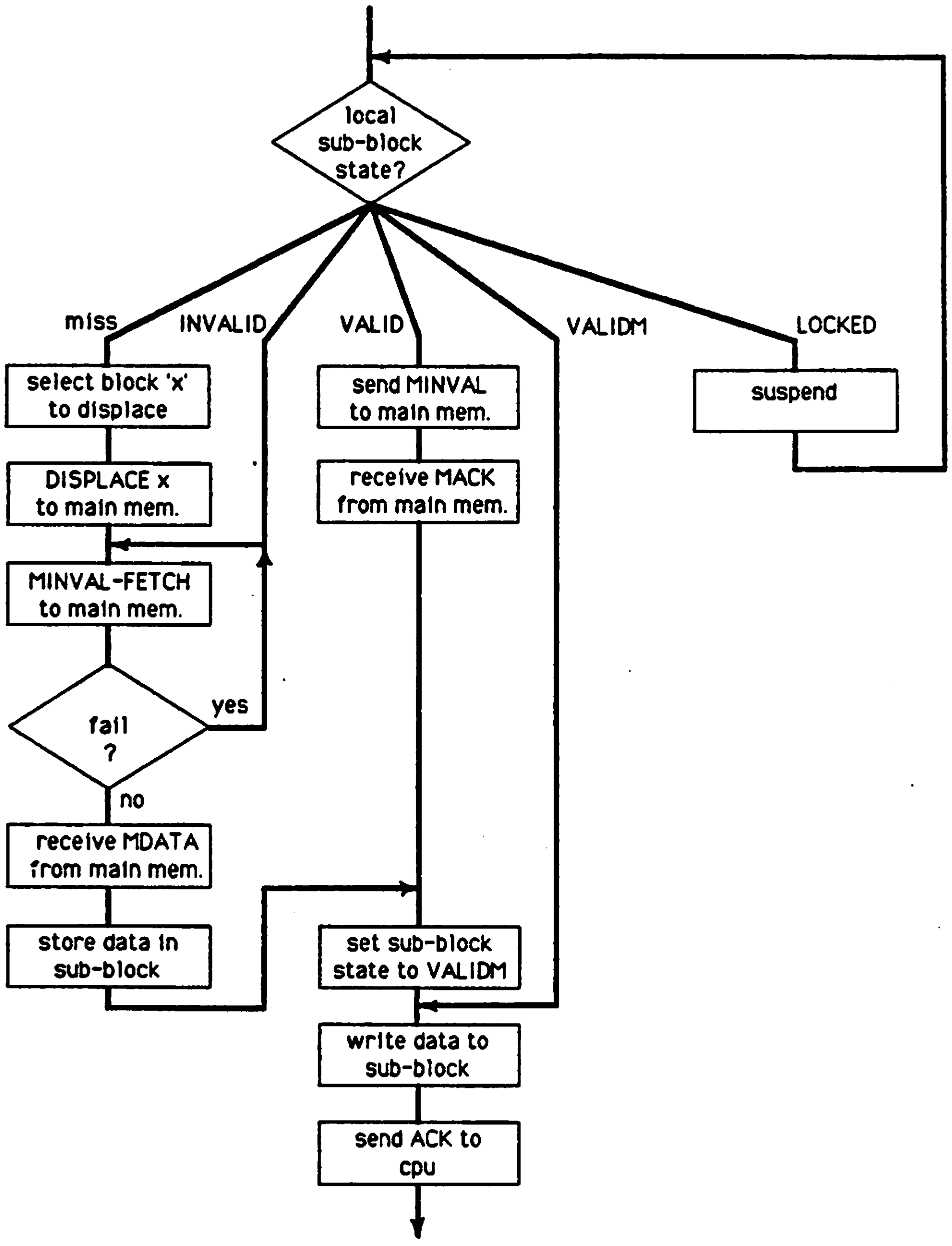

Figure 2.6: Cache Write

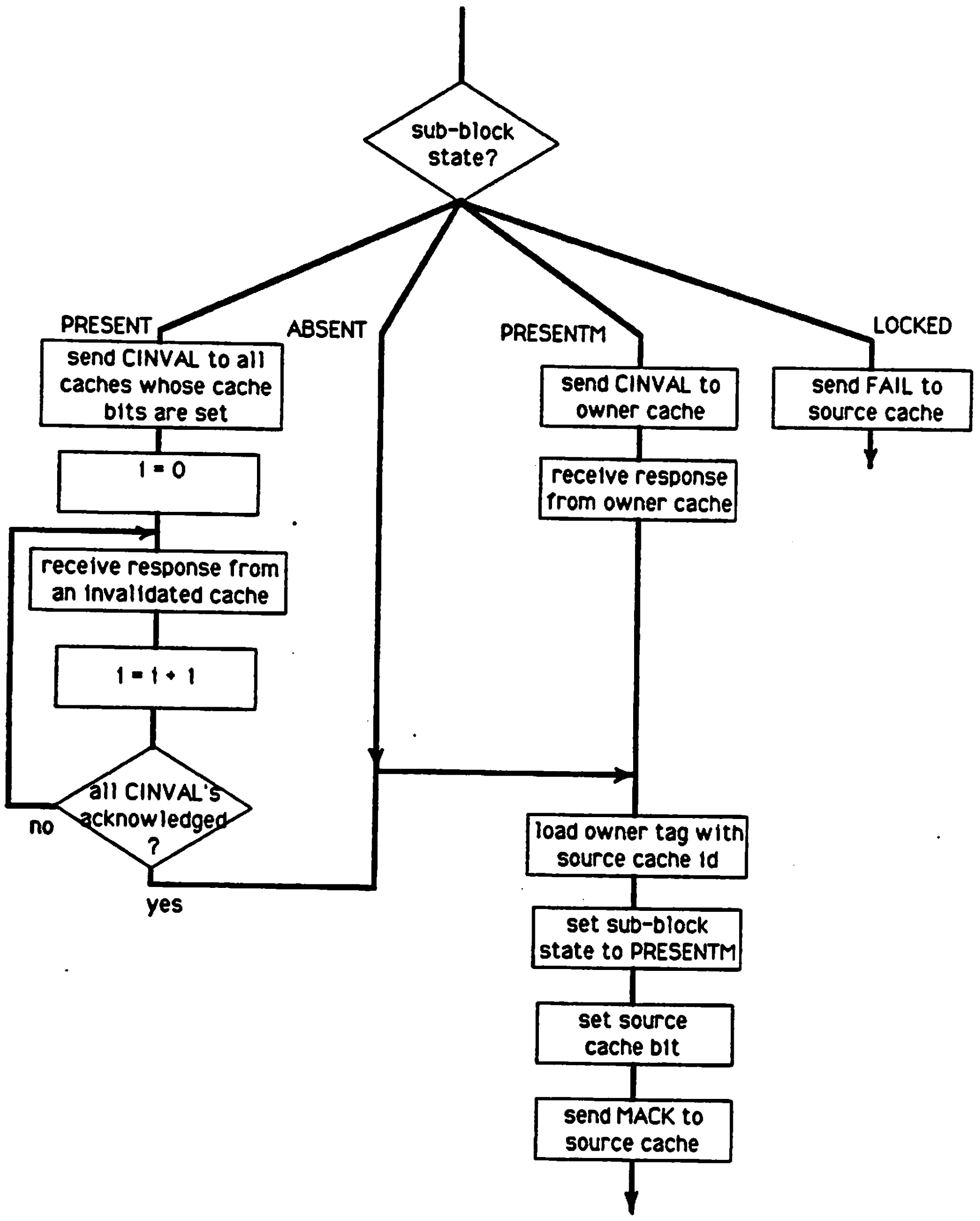
sub-block state?
PRESENT
ABSENT
PRESENTM
LOCKED
send CINVAL to all caches whose cache bits are set
i = 0
receive response from an invalidated cache
i = i + 1
all CINVAL's acknowledged ?
no
yes
send CINVAL to owner cache
receive response from owner cache
send FAIL to source cache
load owner tag with source cache id
set sub-block state to PRESENTM
set source cache bit
send MACK to source cache

Figure 2.7: Memory Invalidate

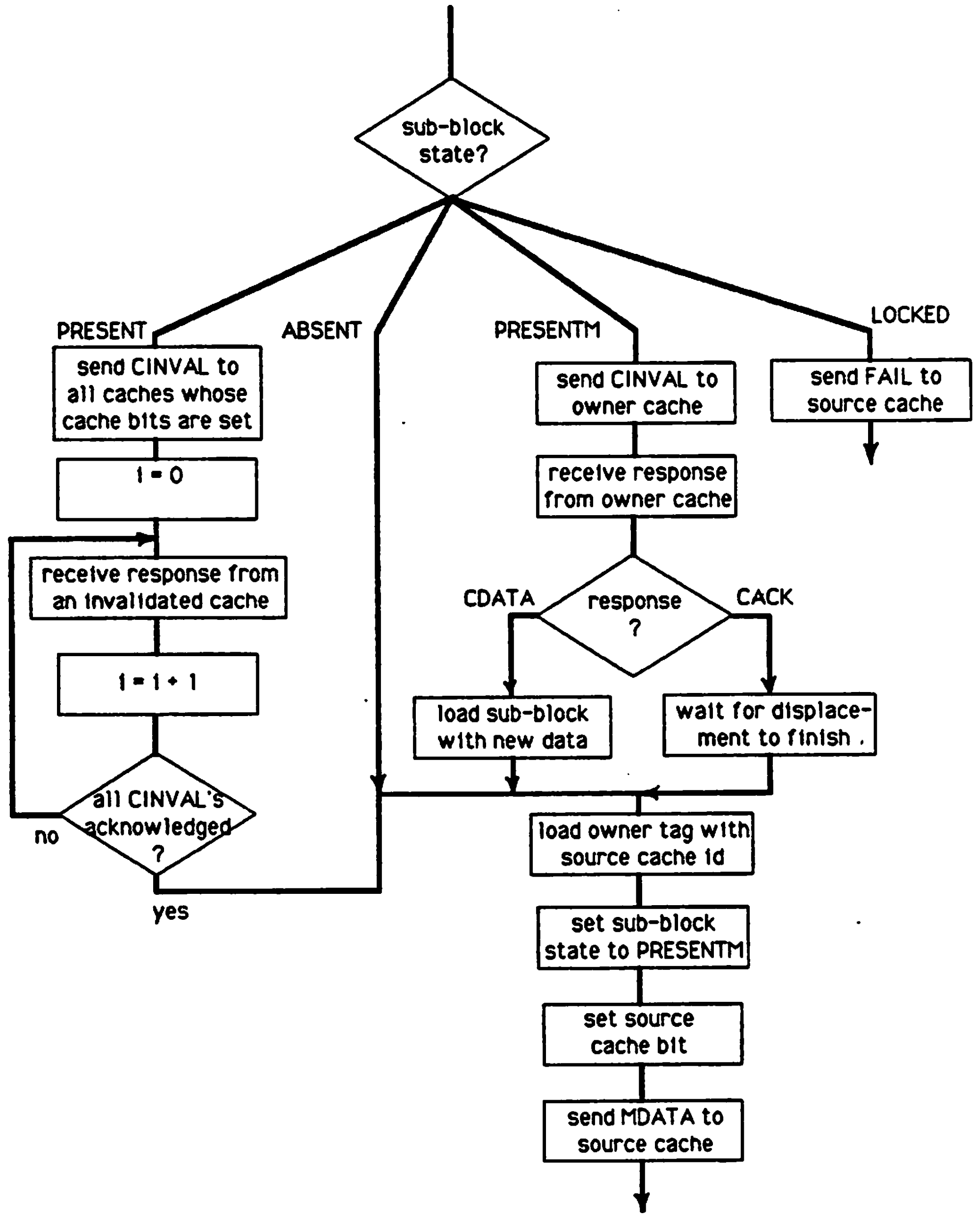
sub-block state?
PRESENT
ABSENT
PRESENTM
LOCKED
send CINVAL to all caches whose cache bits are set
i = 0
receive response from an invalidated cache
i = i + 1
all CINVAL's acknowledged ?
no
yes
send CINVAL to owner cache
receive response from owner cache
response ?
CDATA
CACK
load sub-block with new data
wait for displacement to finish .
send FAIL to source cache
load owner tag with source cache id
set sub-block state to PRESENTM
set source cache bit
send MDATA to source cache

Figure 2.8: Memory Invalidate-and-Fetch

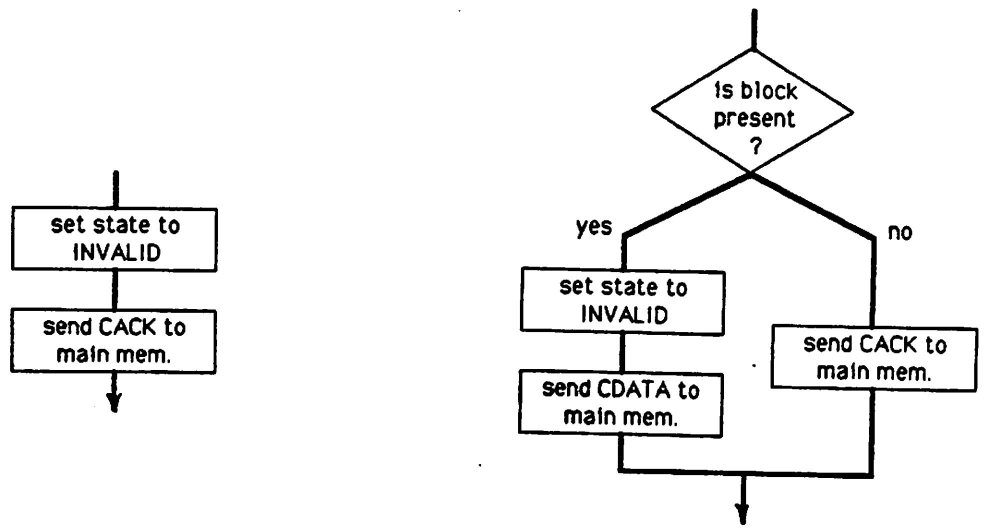

Figure 2.9: Cache Invalidate and Invalidate-and-Fetch

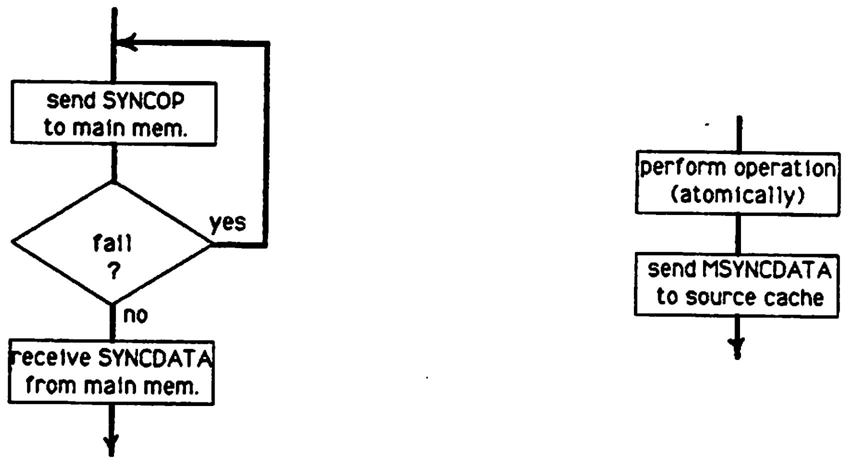
send SYNCOP to main mem.
fail ?
yes
no
receive SYNCDATA from main mem.
perform operation (atomically)
send MSYNCDATA to source cache

Figure 2.10: Cache and Memory Synchronization Operations

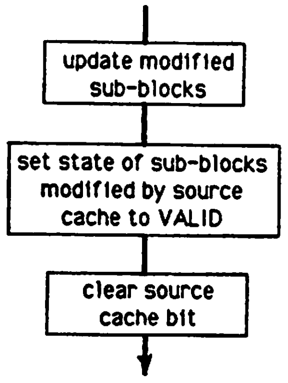
update modified sub-blocks
set state of sub-blocks modified by source cache to VALID
clear source cache bit

Figure 2.11: Memory Displace

a 101 bit tag; this dictates a block size much larger than 16 bytes if the tag is to be much smaller than the data it is associated with. Systems built using this consistency scheme are not easily expanded because the tag length is dependent on the number of processors. Several variations of the basic protocol have been suggested for reducing the tag size. In one variation [AB84], the array of cache bits is eliminated and an extra bit is used to indicate if one or more caches have copies. The protocol in Figures 2.3 to 2.11 is modified so that whenever a block must be invalidated from the main memory a broadcast is used. The communication requirements of this variation can be reduced at the expense of a slightly larger tag by adding a small number, say i, of multiprocessor identifiers which would hold the addresses of caches holding copies of the associated block [A*88]. If the number of copies ever exceeds i, broadcasting is used. This scheme would work well if the average number of copies of a block are low. The results in Chapter 4, however, indicate that the average number of copies is quite high for the benchmarks examined. A slight variation of this limits the maximum number of copies of a block to i. In the following section, propose an alternative way to reduce tag overhead by sectoring a block into sub-blocks is proposed.

2.2.2 Modified Censier and Feautrier Scheme

Another way to reduce tag overhead is to increase the block size but maintain finer granularity for invalidations by dividing blocks into sub-blocks. Finer granularity is needed to reduce the number of cache misses caused by the invalidation of a large block because of a write to a small portion of it. It also allows a block to be referenced without transferring its entire contents. This idea is similar to sectored cache schemes for uniprocessor memory systems [HS84].

With sectored blocks, the Censier and Feautrier protocol is modified as follows. In the main memory directory, each sub-block is given a tag large enough to hold a processor number, plus a modified bit. The array of cache bits is associated with an entire block. A cache bit set to one now indicates that the associated cache holds one or more valid sub-blocks. In the cache directories, the valid and modified bits are now distributed among the sub-blocks. Figure 2.12 illustrates the new tag scheme.

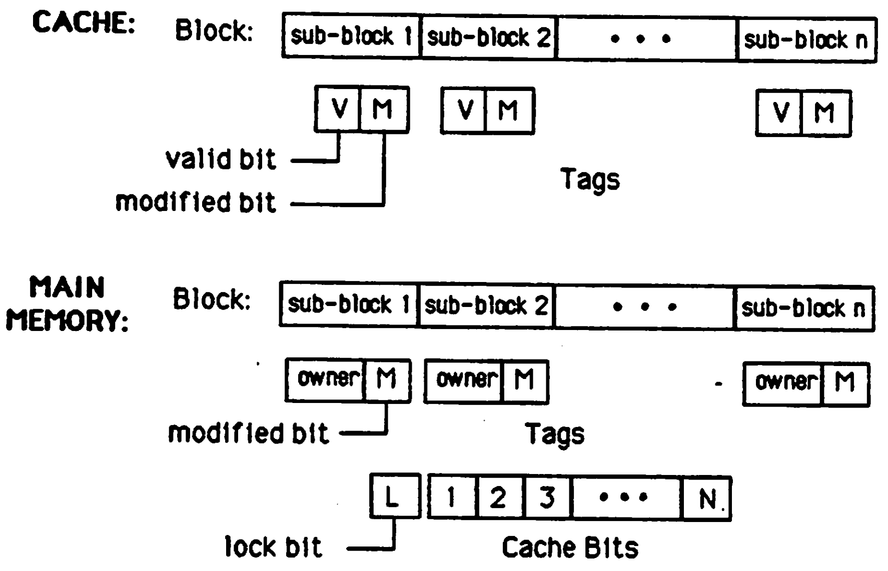

Figure 2.12: Tag Scheme for Modified Censier and Feautrier Protocol

As before, different protocol actions are taken depending on sub-block state in the caches and at the main memory. In the caches, sub-blocks can be VALID, VALIDM or INVALID, depending on their respective valid and modified bits in the same way that cache block state is determined in the original scheme. At the main memory, the state of a sub-block is determined as follows:

1. ABSENT: no caches hold any portion of the block containing the sub-block (all cache bits for the containing block are zero; lock bit is 0);

2. PRESENT: one or more caches hold portions of the block containing the sub-block (one or more cache bits for the containing block are 1, and the modified bit for the sub-block is 0; lock bit is 0);

3. PRESENTM: exactly one cache holds a valid copy of the sub-block, and it is modified (one or more cache bits for the containing block are 1, the modified bit for the sub-block is 1, and its "owner" tag holds the address of the cache holding the modified copy; lock bit is 0);

4. LOCKED: an operation on this block is currently in progress (lock bit is 1);

Protocol actions as a function of cache and main memory state are the same as those for the original protocol (Figures 2.3 to 2.11), with only a minor modification to the way main memory state is updated with a displacement: in the new scheme all sub-blocks must be checked so that those which are PRESENTM and "owned" by the displacing cache are correctly updated.

[HS84] investigated the use of sectored blocks in the context of small, microprocessor on-chip caches and found that sectoring substantially increased the number of cache misses. In the study reported here, however, the cache sizes are much larger and it was felt that misses resulting from accesses to synchronization variables would dominate any extra misses due to sectoring. The results in Section 4 support this reasoning.

The original Censier and Feautrier protocol is clearly a subset of this scheme in which each block has only one sub-block. The original Censier and Feautrier scheme requires N bits per tag per sub-block, where N is the number of processors. In the sectored scheme, $(N+1)/n + \log N + 1$ bits per tag per sub-block are needed, where n is the number of sub-blocks in a block. For $N = 512$ and $n = 32$, the original scheme requires 512 bits per tag while the sectored scheme only requires 26. Tag size can be further reduced by a factor of 2 to 10 if processors are clustered onto busses so that only cluster addresses are needed. While the modified scheme requires less memory for tags, it requires slightly higher communication overhead because invalidations are sent to all caches holding portions of the affected block, some of which may not hold valid copies of the affected sub-block. It was assumed that this additional overhead would be negligible, and simulation results confirmed it. Although the tags in the new scheme are still dependent on the number of processors in the system, the reduced tag overhead should make it possible to build a machine with tag sizes allocated for the worst case (maximum) number of processors.

2.3 Another Extension to Censier and Feautrier's Scheme

The main memory of a large multiprocessor is typically interleaved to provide adequate bandwidth [P*85,G*83a,G*83b]. If the number of main memory banks is made equal to the number of processors and each bank located next to a processor to be used as a second level cache, performance could potentially be improved since the size of such a bank would be 2 to 10 times the size of the first level cache. Some multiprocessors, such as the BBN Butterfly [BC86] and the IBM RP3 [P*85], distribute the main memory among the processors, but none provide hardware support to use the local portions of main memory as second level caches. This section presents a way to further extend Censier and Feautrier's protocol so that it uses distributed main memory banks as second level caches. In the following discussion a multiprocessor organization with distributed main memory (Figure 2.13) is assumed, with a third level added to the memory hierarchy: the *backup* memory. The backup memory is used to hold blocks that do not fit in the second level. Since the backup memory performs a function similar to that of a paging device in a virtual memory system [Den70], we expect this level of the hierarchy to be accessed relatively infrequently. For the remainder of this section a reference to a cache means the second level cache of Figure 2.13; the first level cache is neglected, assuming that it is managed using extensions of uniprocessor cache techniques, and that its net effect is to reduce the access time to the second level cache.

If main memory is broken up into a set of caches, data can no longer be statically assigned to main memory banks as was assumed in the consistency schemes of Sections 2.1 and 2.2.2. These schemes, and virtually all other proposed cache consistency methods, assume that all cache blocks have statically determined home locations in the main memory so that the caches always know where to fetch from or displace to. The differences then, between the extended Censier and Feautrier protocol and the original are:

1. Caches now must act as the main memory would in the original scheme for those blocks for which they are "owner."

2. Caches must be capable of finding a block's owner dynamically, since it can vary throughout program execution.

In the extended protocol each cache block is given a set of tags which is the union of those provided for cache blocks and main memory blocks in Section 2.2.2 (Figure 2.14).

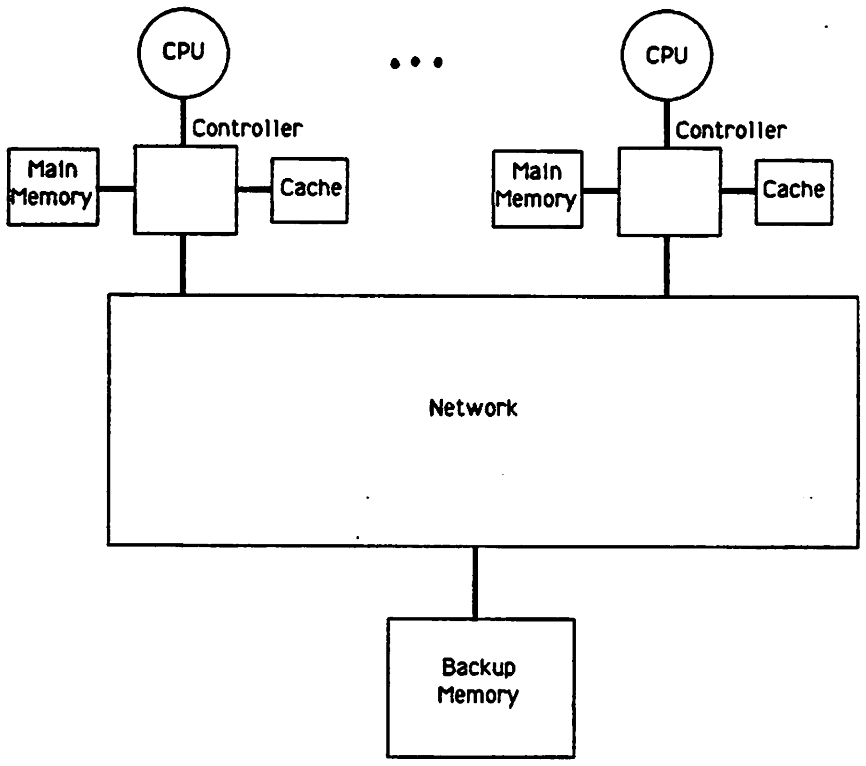

Figure 2.13: Multiprocessor Architecture with Distributed Main Memory

In addition, an "owner" bit and owner tag are associated with each block: the bit indicates if the cache is the owner of the block; the tag holds the address of the "likely" owner of the block if the owner bit is 0. There is only one owner of a block in the entire system. The tags which were provided for a main memory block in Section 2.2.2 are used by a cache only if it is not the owner of the associated block. When used, each respective set of tags have the same meaning as in Section 2.2.2: if a cache owns the containing block (owner bit is

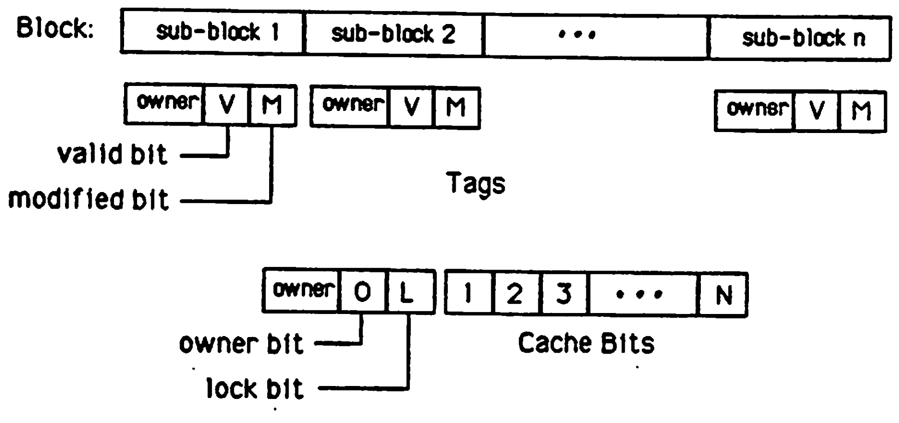

Figure 2.14: New Tag Set

set), a sub-block can be LOCKED, ABSENT, PRESENT or PRESENTM, with meanings exactly as before; similarly, if a cache does not own the containing block, a sub-block can be INVALID, VALID or VALIDM.

The network transactions used by the new protocol include those used in Section 2.2.2, with one addition and one modification. The additional transaction is:

REFUSED(): indicates that the cache to which a transaction was issued cannot satisfy it because it is not the owner.

The modification is the addition of an extra parameter to the DISPLACE transaction as follows:

DISPLACE(*addr, data, ownerflag*): where *ownerflag* is 1 if the displacing cache owns the block, and 0 otherwise.

The actions performed by the cache controller are the same as those in Figures 2.3 to 2.11 with the following four changes:

1. The "main memory" actions (MRead, etc.) are now performed by the caches. The

flowcharts in Figure 2.3 to 2.11 must have the test shown in Figure 2.15 appended to the top. This test simply determines whether the cache receiving the transaction holds the referenced block, and if so, if it is the owner. The cache can only handle the transaction if it is the owner, otherwise it returns a REFUSED() message.

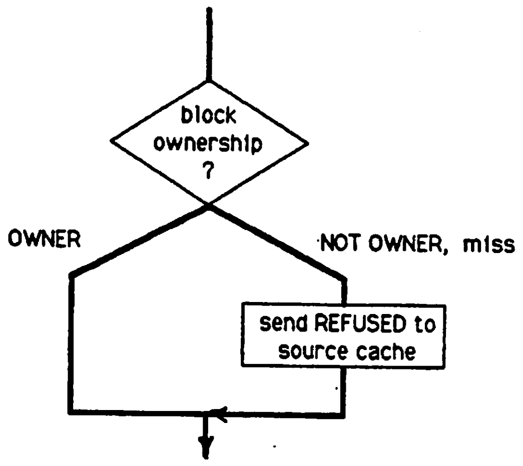

Figure 2.15: Ownership Check

2. Transactions directed to main memory (MREAD, MINVAL, MINVAL-FETCH or MSYNCOP) must be altered to search for the cache serving as "main memory" for the affected block. Figure 2.16 shows the procedure that replaces the appropriate parts of Figures 2.3 to 2.4. These more complex procedures do the following. If the cache issuing the main memory transaction owns the affected block, a network access is avoided and the transaction is handled locally. Otherwise, the cache begins its search by sending the transaction to the likely owner of the block, as indicated in the block owner tag. If the likely owner responds with failure, the transaction is retried. If the likely owner accepts the transaction, the procedure continues with the appropriate steps in Figures 2.3 to 2.11. If the transaction is refused, the issuing cache broadcasts the transactions and collects all of the responses. Since only one cache can be an owner, at most one response contains data or an acknowledgement, and it is buffered if it arrives. If any cache fail to receive the transaction because of a locked block, the transaction is re-issued to that cache. If all caches refuse to accept the transaction, the backup memory is checked. If it also refuses to accept the transaction, the broadcast is repeated again. An owning cache can be missed during

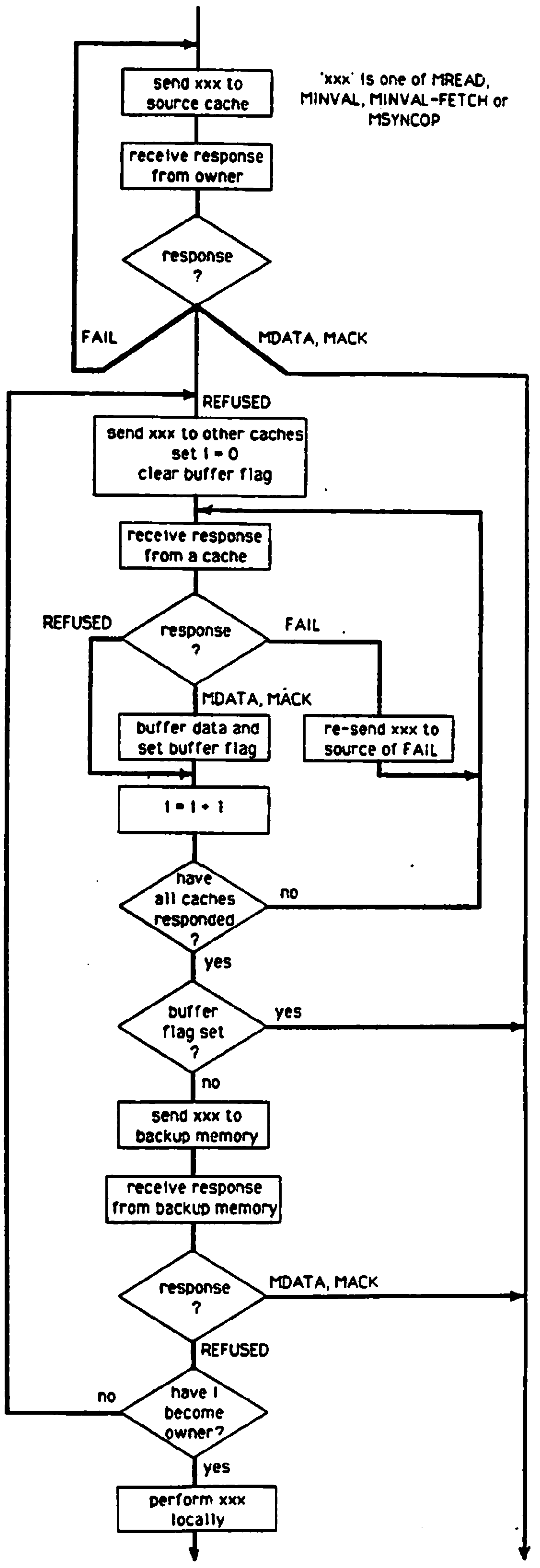
send xxx to source cache
'xxx' is one of MREAD, MINVAL, MINVAL-FETCH or MSYNCOP
receive response from owner
response ?
FAIL
MDATA, MACK
REFUSED
send xxx to other caches set i = 0 clear buffer flag
receive response from a cache
REFUSED
response ?
FAIL
MDATA, MACK
buffer data and set buffer flag
re-send xxx to source of FAIL
i = i + 1
have all caches responded ?
no
yes
buffer flag set ?
yes
no
send xxx to backup memory
receive response from backup memory
response ?
MDATA, MACK
REFUSED
no
have I become owner?
yes
perform xxx locally

Figure 2.16: Modified Search Procedure

a broadcast if the network delays are such that an unsearched cache can displace the ownership of the affected page to an already searched cache. [1] The search loop should eventually succeed unless the caches displace pages at an extraordinarily high rate. [2]

3. Caches displacing blocks must search for a cache that can accept the displacement. Figure 2.17 shows the new displacement procedure which replaces those steps of Figures 2.3 to 2.11 labelled "DISPLACE x to main mem." As in 2, the search is dependent on the block status in the displacing cache. If the cache owns the block and it is shared, displacement transactions are sent one at a time to other caches holding copies; if a particular cache refuses the displacement, the next cache with a copy is tried. A cache with a copy will only refuse a displacement if it displaces its copy just before it receives the displacement transaction. If none of the caches with copies accept the displacement, the block must now be owned and unshared, so the entire set of caches must be tried serially. If none of these accepts the displacement, the backup memory, which always accepts displacements, is used. If a cache displaces an unowned block, it first tries to displace to the likely owner, as indicated by the block owner tag. If the likely owner refuses, it reverts to a serial search of all caches as above. If the serial search fails in the unshared case, however, the displacing cache must check to see if it has recently assumed ownership because of the owner performing a displacement in the middle of the search. [3] If the displacing cache is not the owner, it must repeat the serial search because some cache in the system still has ownership of the block (because a cache cannot displace to the backup memory unless it has the sole copy of the affected block).

4. The procedure for DISPLACE in Figure 2.11 is replaced by that of Figure 2.18. In the new protocol a cache can accept a displaced block only if:

   (a) the accepting cache has a free block and the displaced block is being displaced by its owner;

   (b) the accepting cache has a copy of the block, and either the accepting cache or

[1] While this might seem like a rare situation that might never occur in practice, the simulations proved otherwise.

[2] It is assumed that the caches use a least recently used (LRU) replacement scheme, and that when a cache accepts a displaced block it updates its LRU status.

[3] It is possible for the owner to be missed during a serial search if the owner makes a displacement, before it is searched, to a cache which has already been searched.

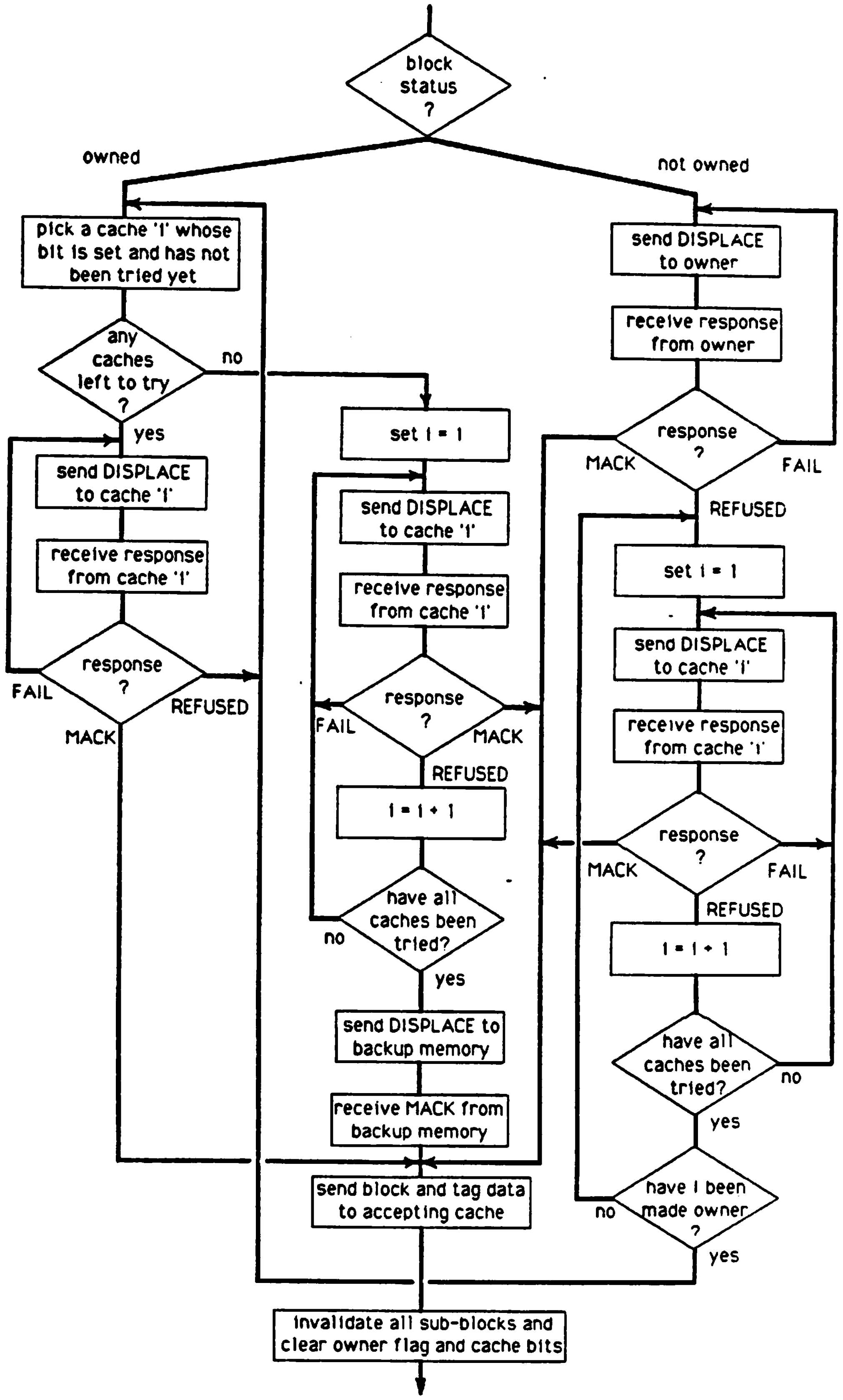
block status ?
owned
not owned
pick a cache 'i' whose bit is set and has not been tried yet
any caches left to try ?
no
yes
send DISPLACE to cache 'i'
receive response from cache 'i'
response ?
FAIL
MACK
REFUSED
set i = 1
send DISPLACE to cache 'i'
receive response from cache 'i'
response ?
FAIL
MACK
REFUSED
i = i + 1
have all caches been tried?
no
yes
send DISPLACE to backup memory
receive MACK from backup memory
send DISPLACE to owner
receive response from owner
response ?
MACK
FAIL
REFUSED
set i = 1
send DISPLACE to cache 'i'
receive response from cache 'i'
response ?
MACK
FAIL
REFUSED
i = i + 1
have all caches been tried?
no
yes
have i been made owner ?
no
yes
send block and tag data to accepting cache
invalidate all sub-blocks and clear owner flag and cache bits

Figure 2.17: Modified Displacement Search

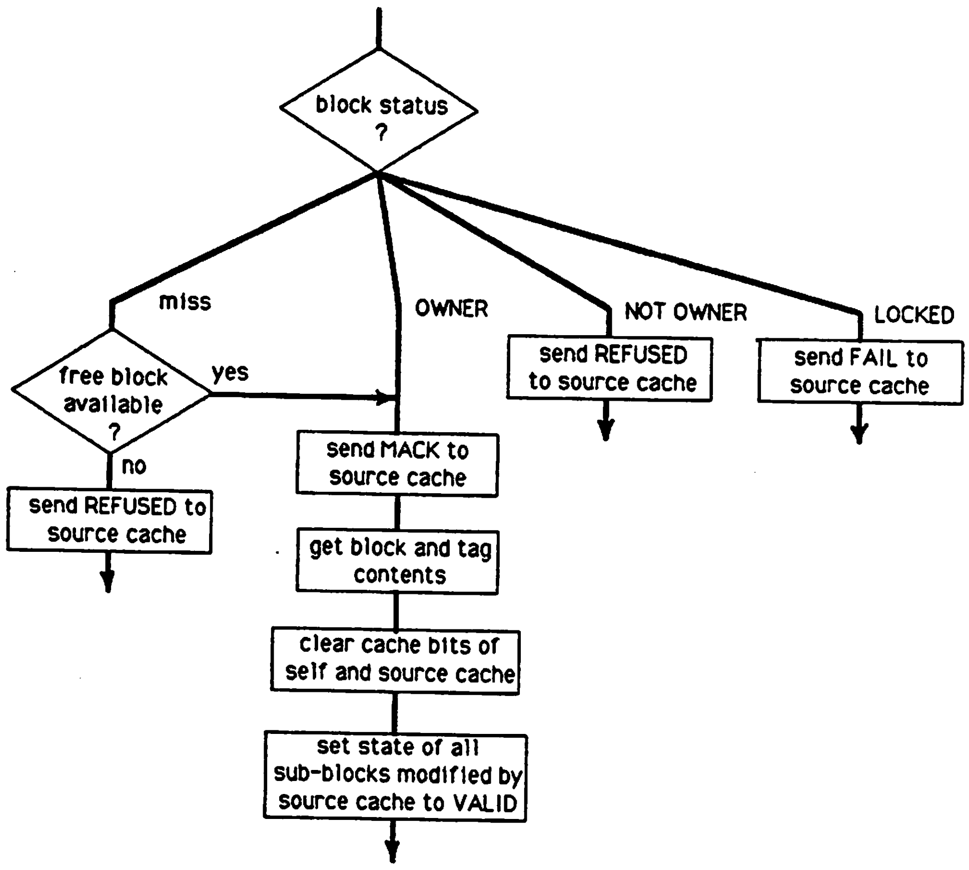
block status ?
miss
OWNER
NOT OWNER
LOCKED
free block available ?
yes
no
send REFUSED to source cache
send MACK to source cache
get block and tag contents
clear cache bits of self and source cache
set state of all sub-blocks modified by source cache to VALID
send REFUSED to source cache
send FAIL to source cache

Figure 2.18: Modified Displacement Procedure

the displacing cache is an owner.

These restrictions ensure that the owning cache is always involved in a displacement so that the owner's set of tags remains consistent. For example, if a cache with an unowned copy of a block could displace to another cache with an unowned copy the cache bits held by the owner would not be updated to show that the displacing cache no longer has a copy.

Clearly, this new protocol is considerably more complex than the protocol of Section 2.2.2, and in the worst case most transactions take much longer to perform. It was assumed, however, that in practice the worst case would occur infrequently and that most transactions could be handled in about the same time as in the basic protocol; most of the time the controllers should take actions identical to those of the sectored Censier and Feautrier scheme. The simulation results in Chapter 4 support this. If the complex portions of Figures 2.15 to 2.18 are sufficiently rare, they can be implemented in software.

2.4 Tag Overhead

In this section it is shown that both modified versions of Censier and Feautrier's directory method consume an amount of memory for tags that is a reasonably small fraction of that used to hold data. Examination of Figures 2.12 and 2.14 shows that the two modified schemes require the following amount of memory for tags:

Modification 1 (Figure 2.12)

Main Memory:

- N cache bits per block, where N is the number of processors in the entire multiprocessor;
- b owner tags of size $\log_2 N$ bits each, where b is the number of sub-blocks per block;
- b modified bits per block;
- 1 lock bit per block;

Cache:

- 1 modified bit per sub-block;
- 1 valid bit per sub-block;

Modification 2 (Figure 2.14)

Main Memory/Cache:

- N cache bits per block;
- b owner tags of size $\log_2 N$ bits each;
- b modified bits per block;
- b valid bits per block;
- 1 lock bit per block;
- 1 owner bit per block;
- 1 likely owner tag, of size $\log_2 N$ bits each, per block;

Tag sizes T_1 and T_2 for variations 1 and 2 are thus:

$$T_1 = N + b(\log_2 N + 1) + 1$$

$$T_2 = N + b(\log_2 N + 2) + 3$$

Figures 2.19 and 2.20 show plots of $\frac{T_1}{\text{blocksize}}$ and $\frac{T_2}{\text{blocksize}}$ versus block size with sub-block size as a parameter and $N = 512$. The plots show that tag overhead is similar for both modifications for block sizes of 16 or greater, and is less than 15% for a sub-block size of 16 bytes, page size of 1024 bytes, and 512 or fewer processors.

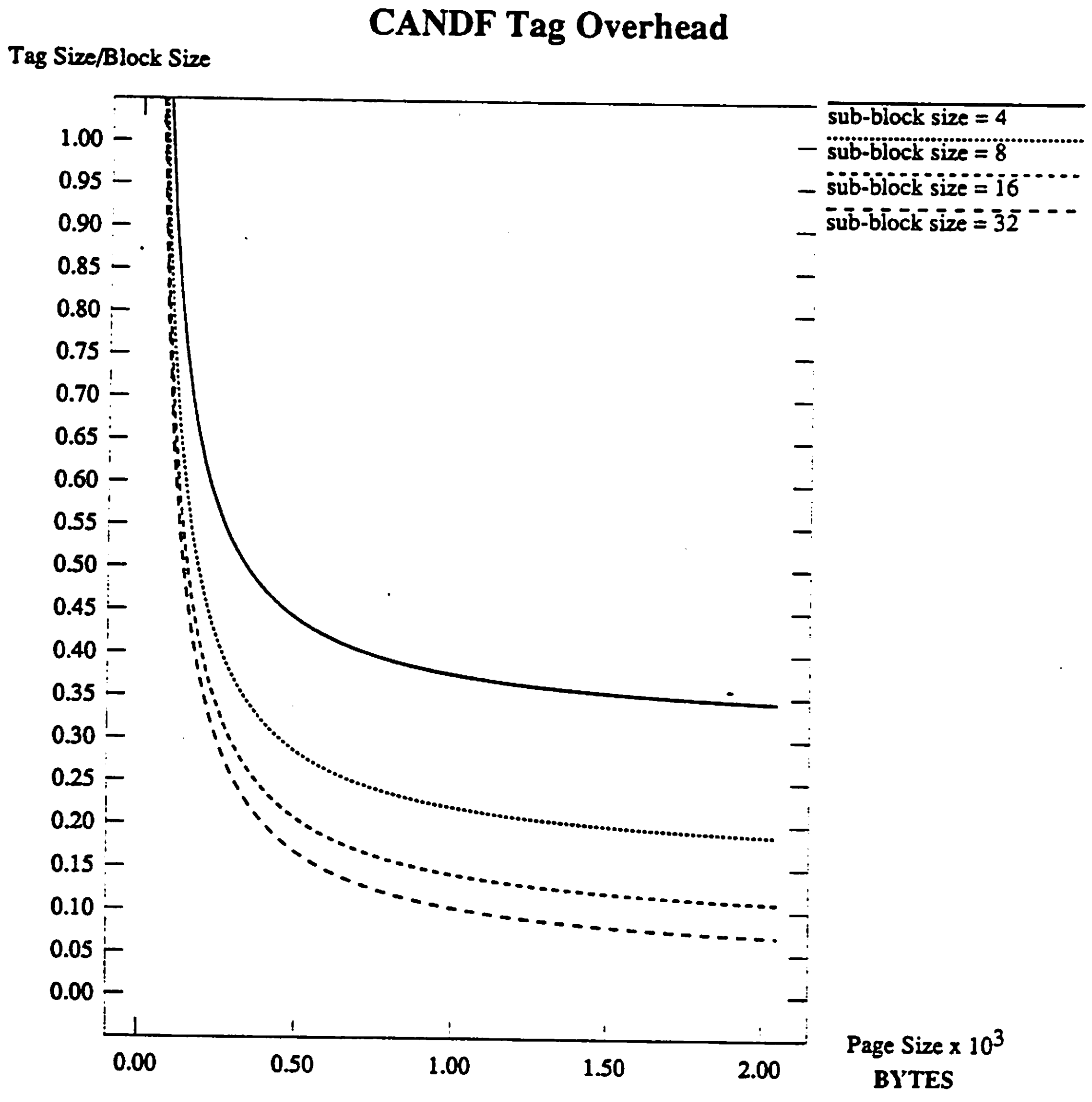

Figure 2.19: Tag Overhead for Variation 1 with 512 Processors

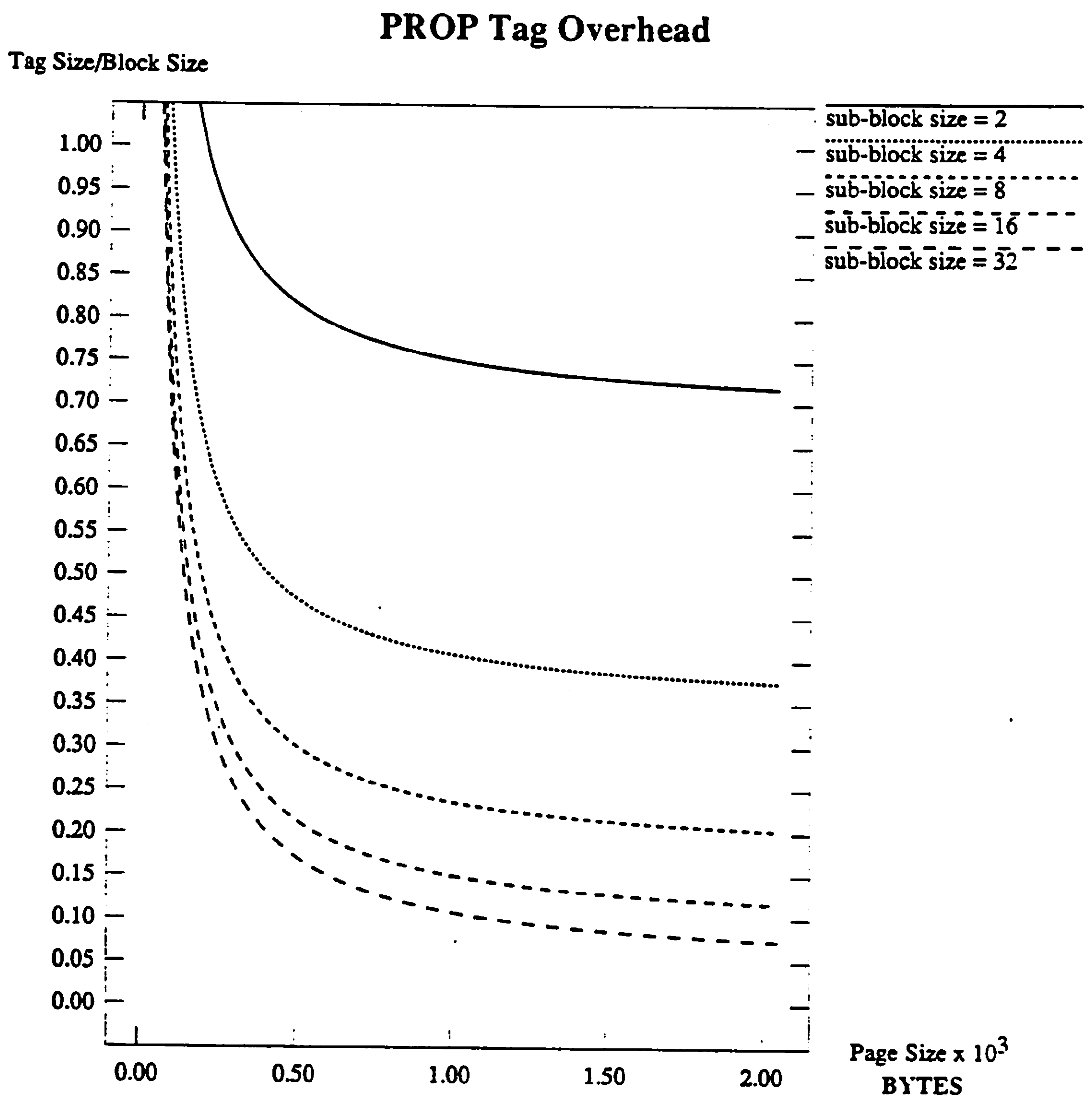

Figure 2.20: Tag Overhead for Variation 2 with 512 Processors

Chapter 3

Evaluation Methodology

The three consistency methods of Chapter 2 were evaluated using simulations of a set of benchmark programs. In this chapter the methods, assumptions, and goals of the evaluation are described. Evaluation results are presented in Chapter 4. This study uses two figures of merit: average communication requirements and average memory access time. Average communication requirements are presented as the average number of normalized network transactions issued by a single processor per memory reference. A normalized network transaction is simply a transaction involving one word (4 bytes) of data, from which it is assumed more complicated transactions can be expressed as multiples. It is assumed that transactions that require no transfer of data have the same network overhead as those transferring one word. Normalized network transactions reflect the effects of cache miss ratios and volume of transmitted data. In like manner, average memory access time is presented as the average number of normalized time units required to satisfy a single memory reference, to data or instructions, by one processor. Here a normalized time unit is the time required to reference data in a local cache. The simulation study was made with the unrealistic assumption that network delay is independent of network load. This assumption simplifies the analysis considerably, and when two consistency schemes have similar communications requirements it should not prevent a valid comparison of average memory access times. If, however, average access time is reduced at the expense of substantially increased network load, the cost of a higher performance network must be considered.

For the rest of this report the names of the cache consistency schemes are abbreviated as follows:

1. NOCACHE: shared writeable data is not cached;

2. CANDF: Censier and Feautrier scheme with sub-blocks;

3. PROP: proposed scheme in which main memory is used as a second level cache;

Most simulation studies of computer memory systems use address traces collected from real machines running a set of benchmark programs [Smi82,Sto87]. Of these studies, most of those considering multiprocessors have dealt with relatively few processors, typically under 10 [A*88,A*84,EK88]. For large numbers of processors, trace driven methods become cumbersome because of the large amount of storage required to hold the traces. As an example, traces for 64 processors with one million 32 bit references per processor require 256 megabytes of storage in uncompressed form. The motivation for trace driven simulation is also reduced because multiprocessor memory systems require more complex simulation models, thus relatively little extra simulation time is required to simulate the processors executing the code. For these reasons, a simulator was developed to perform simulations of complete multiprocessors in sufficient detail to evaluate the performance of different memory systems. Although considerable effort was required to develop the simulator, and simulations take somewhat longer than when traces are used, it requires much less storage and does not depend on using an existing machine to collect traces.

The simulation study involved simulations of four benchmark programs on three multiprocessor architectures supporting the cache coherency techniques of the preceding chapter. As well as providing average access times and network traffic directly, the simulator also gathered other statistics, including the average number of copies of shared blocks and the average number of invalidations required per shared write. Each simulation required the following steps:

1. Compile the shared memory benchmark program into 68020/68881 assembler using a commercial C compiler. The *HPUX cc* compiler available on Hewlett Packard series 350 workstations was used.

2. Convert the assembly code into the pre-decoded format required by the simulator. A special assembler was written to do this, and several assumptions about the virtual address space were made. A global addressing space with a 32 bit address of form shown in Figure 3.1 was used. In a global address space, to each piece of logical piece of

data there corresponds exactly one virtual address; if two or more processes reference the same data they must use identical virtual addresses. This type of address space

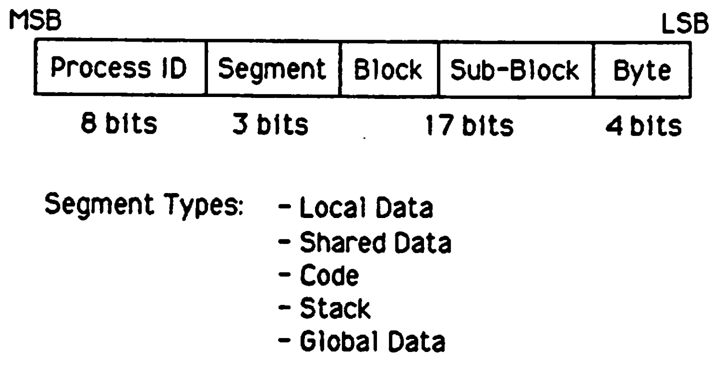

Figure 3.1: Components of an Address

was chosen to avoid the problems associated with "synonyms"—two or more distinct virtual addresses corresponding to the same data [Smi82].

As Figure 3.1 shows, the address space is segmented with each segment uniquely identified by a segment identifier and segment type. The segment type field is used by the memory controllers to determine the shared writeable status of each CPU reference, as well as in checking for access violations.

3. Prepare a netlist of the architecture to be simulated, providing specifications for and interconnections among a set of CPU's, memory controllers and main memories.

4. Prepare initialization files for each simulation model, providing initial memory and register contents, delay information and address mapping data. These files were generated by a program using the output of Step 2 and a list of processors with the programs they were to run.

5. Run the simulation.

6. Post-process the results.

In this chapter the simulator, the simulation models and the benchmark programs are described in detail. The operating system support required for the simulations is also described, along with details of the three multiprocessor architectures whose simulation results are presented in Chapter 4.

3.1 The Simulator and Simulation Models

The multiprocessor simulator was constructed in a modular fashion (Figure 3.2), with a core providing message passing facilities for a simulator shell and collection of architectural models. The models include a 68020/68881 CPU, three different memory controllers, two main memories and a statistics gathering module. The following subsections describe each of the simulator components in greater detail.

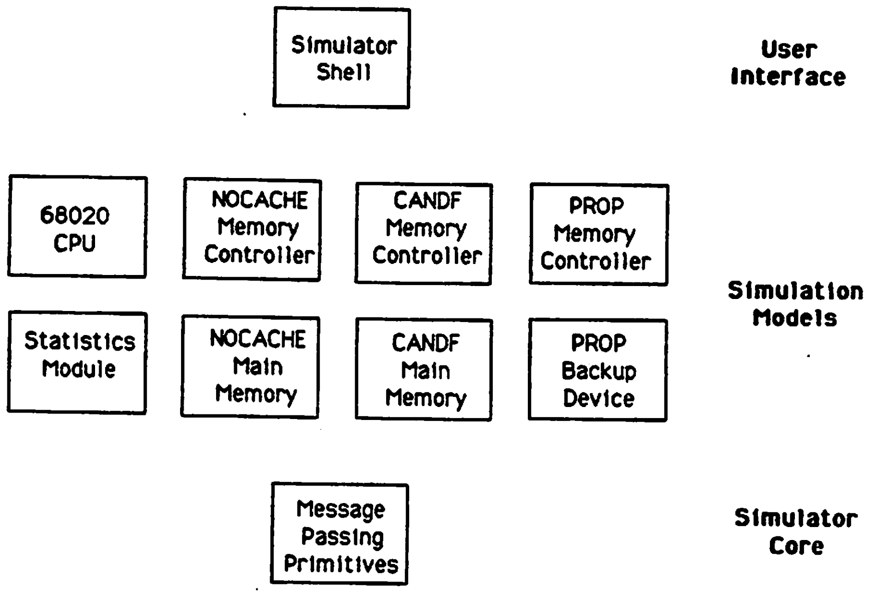

Figure 3.2: Simulator Components

No network model is shown in Figure 3.2 because a simple network model was incorporated into the controller and main memory modules. In the simple network model it is assumed that:

- Each processor is assigned a unique identifier between 0 and $N-1$, inclusive, where N is the number of processors.
- N is a power of 2.
- For the NOCACHE and CANDF schemes, memory is interleaved N ways and distributed among the processors so that bank i is associated with processor i.
- For the NOCACHE and CANDF schemes, interleaving is done on the lower order

$\log_2 N$ address bits of the block address field (Figure 3.1). (ie. if the lower order bits of a block address are j, main memory bank j contains the referenced data).

- Inter-network delays depend only on the processor identifiers of the source and destination of a network transaction.

With these assumptions, the controller and memory modules account for network delay by simply looking up a delay value in a two dimensional table indexed by source and destination identifiers. Since the delay table was made a simulation parameter, different networks can be modelled to first order accuracy by using different delay tables. In the simulations the following two network models were used:

Uniform: The delay between any two distinct processors is constant (local references are modelled by a separate delay parameter).

Cube: The delay between any two processors is the Hamming distance between the source and destination identifiers, multiplied by a common scale factor.

3.1.1 Message Passing Core

The message passing core provides a set of functions that operate on *messages*, which are characterized by a *message type* and the *data* which they contain. The data passed in a message can be anything expressible as a structure in the C programming language, the language in which the simulator is written. Messages are created and sent between instances of simulator models, which are comprised of (Figure 3.3):

1. *Input ports*, from which messages are received from other model instances.
2. *Output ports*, from which messages are sent to other model instances.
3. Sets of message types, one set per port, identifying messages which may be received/sent by the respective input/output port. The sets of messages flowing through a port cannot change during a simulation. These sets are common to all instances of the same model.
4. *Message handlers*, at least one for each message type in 3 associated with an input port. The message handlers specify the actions to perform for each type of message

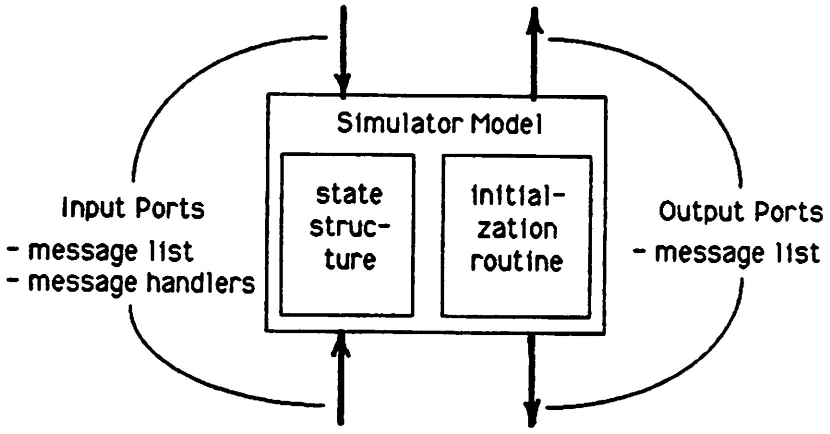

Figure 3.3: Simulator Model

that is received. When a handler is invoked, it is given the contents of its corresponding message, the current simulation time and a pointer to the state structure (see 5) of the model instance to which the message has been sent. The message passing core provides the following two primitives which may be used in writing message handlers:

- SEND(*t*, *destination*, *type*, *data*): send a message of type *type* containing *data* to *destination* at time *t*.
- SET_HANDLER(*type*, *function*): change the handler for messages of type *type* to be *function*.

A collection of message handlers is common to all instances of the same model.

5. A state data structure, unique to each model instance, to preserve information between the receipt of messages.

6. An initialization routine, invoked at the beginning of a simulation. The initialization routine may be passed a list of parameters which may be used to initialize the state structure, set delay values and perform other preliminary activities.

With this modelling paradigm a simulation consists of the sending, receiving and processing of messages among a collection of model instances. The message passing core provides the facilities for maintaining simulated time, buffering messages until their sending time is equal to the current simulated time, and invoking the message handlers. The core also provides support for creating instances of models, checking that the sets of messages sent and received by two interconnected ports are the same, and invoking the initialization routines.

The implementation of the message passing core used an event driven algorithm [Ulr78] based on a linked list time queue. In the time queue outstanding messages are stored in a time sorted list of lists (Figure 3.4) where each message list i contains those messages which are to be sent at time t_i. When all of the handlers have been invoked

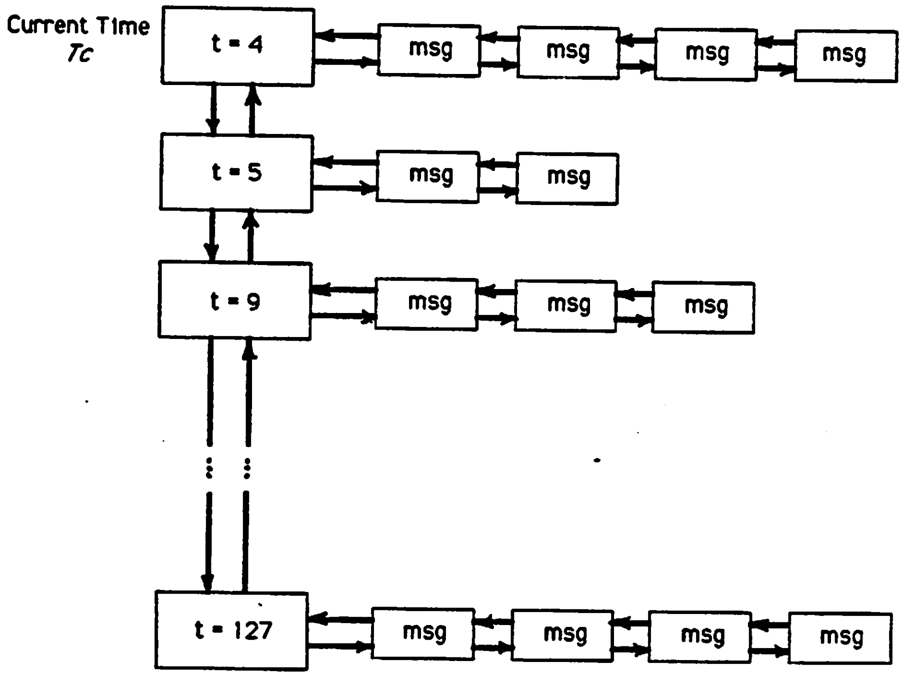

Figure 3.4: Linked List Time Queue

for the recipients of all messages at the current time t_c, t_c is advanced to t_{i+1}, the time associated with the next message list. A simulation terminates when the time queue becomes empty. Although a timing wheel approach may provide better performance [Ulr78], the linked list implementation proved to be satisfactory, permitting approximately two thousand 68020 instructions to be simulated per second of real time on a HP 350 workstation. The simulations were thus about 2000 times slower than the hardware implementation of a single processor.

3.1.2 68020/68881 CPU

The model of the 68020/68881 CPU supports a large subset of the Motorola 68020/68881 instruction set in sufficient detail to permit the assembler output from a C compiler to be simulated correctly with reasonable efficiency. The 68020 architecture was chosen because it is a popular, current design for which commercial compiler support is readily available. The model has the following characteristics:

1. It models the non-pipelined execution of a large subset of the 68020/68881 instruction set. 130 instructions are supported for 9 of the possible addressing modes. For efficiency, data and instructions are treated separately within the simulator so that instructions can be stored in a more easily interpreted format than that defined by Motorola. For additional efficiency, floating point numbers (64 bit only) are stored in a format native to the machine on which the simulator runs. For simplicity, pipelining is not modelled. It is assumed that the effects of pipelining can be modelled, to first order, by reducing the average time to execute an instruction.

2. It has one input port and one output port to communicate with one of the three memory controllers described below.

3. Processing delays are modelled using a the same parameterized delay between each memory reference, regardless of whether an address is being calculated or the instruction is being executed.

4. The processor stalls on writes. Although most high performance processors would not do this, it greatly simplifies the model since complicated buffering and interlocks would be required otherwise. It was assumed that this simplification would have little affect on the relative performance of the three coherence schemes.

5. An assembly level debugger supporting breakpoints, status reporting and tracing is provided.

6. There is full modelling of byte, word, long word and 64 bit floating point operands, including support for 1, 2, 3 and 4 byte alignments.

3.1.3 NOCACHE Memory Controller with Cache

This model simulates the NOCACHE coherence protocol of Section 2.1, and contains a fully associative LRU cache with variable block size. An instance of this model has two pairs of input and output ports: one pair to communicate with a CPU and one pair to communicate with a multiprocessor interconnection network. A single delay is used to model the time between receiving a message from the CPU or network and issuing another one. This delay corresponds to the time to process a cache hit. In this model it was assumed that displacements are buffered so that the controller never has to wait for a displacement to complete.

3.1.4 NOCACHE Main Memory

This model simulates a large interleaved main memory supporting the NOCACHE coherence protocol. Since virtual addresses are sent from the memory controllers it performs virtual to real address translation. A single delay is used to model main memory access time. Interleaving is approximated by neglecting queuing effects (ie. no main memory access is ever delayed waiting for other main memory accesses to complete, with multiple references to the same address excepted). Although this is an optimistic assumption, it was assumed to have little affect on the relative results. If necessary, queuing effects may be modelled to first order by factoring an estimated queuing delay into the main memory access delay. Interactive debugging commands, accessible from the simulator shell described below, support the setting of breakpoints, dumping of data and instructions, and searching instructions for references to a given assembler symbol.

3.1.5 CANDF Memory Controller with Cache

This model is similar to the NOCACHE controller, except it simulates the CANDF protocol of Section 2.2.2. The timing model used is similar to that in the NOCACHE model. It was assumed that in most cases simultaneous CPU and network messages can be handled simultaneously (using, for example, a dual directory technique [BD86]). It was also assumed that the queuing effects of network accesses are negligible. As in the NOCACHE controller, it was assumed that displacements are buffered so that the controller never waits for a displacement to complete.

3.1.6 CANDF Main Memory

The CANDF main memory module models a large interleaved main memory supporting the CANDF coherence protocol. As for the NOCACHE case, virtual to real address translation is performed, a single delay parameter is used, and interleaving is approximated by neglecting queuing effects. In addition, the model permits main memory accesses to be handled even while invalidations or sub-block fetches due to past references are still in progress; as before, this is only permitted for references to distinct addresses. Debugging facilities similar to those in the NOCACHE model are also provided.

3.1.7 PROP Memory Controller with Memory

This model simulates the PROP coherence protocol of Section 2.3, using a block of main memory as a second level cache. As in the NOCACHE and CANDF controllers, two pairs of ports are provided for communication with a CPU and network. The delay model is similar to that used in the NOCACHE and CANDF controllers. This controller model does account for queuing effects on messages from the network, unlike the other controller models. In the NOCACHE and CANDF schemes, main memory is statically interleaved on the lower bits of the block address. It was assumed that this would result in minimal queuing delay at the memory banks, and a simpler simulation model that neglected queuing effects altogether was used. For the PROP scheme, however, it is possible at the beginning of program execution for all data to be clustered in the single memory bank of a parent process. In such a case the queuing effects could be considerable, so they were modelled in the PROP case.

Also unlike the other two controllers, a displacement is completed before satisfying the processor request which caused the displacement. This was done because considerable extra complexity would be required to model buffered displacements for the complex PROP scheme. To reduce the effect of this modelling discrepancy, the results from the NOCACHE and CANDF simulations were post-processed to reflect the additional delay incurred by not buffering displacements.

3.1.8 PROP Backup Memory

This module acts as a single memory bank on which all program blocks reside at startup and on which all dynamically allocated memory is initially assigned. It is also a location where blocks are displaced in the rare event that there is no free space at the second level. It roughly models a third level in the memory hierarchy that could be implemented using disk storage units. For this study, however, the latency of the backup device was the same as that of the main memory. The modelling of disk latencies for the PROP case would have skewed its performance unfairly since disk latencies are not modelled in the NOCACHE and CANDF cases: here it is assumed that at startup all program blocks reside in the main memory. These assumptions are reasonable considering that most results of Section 4 concern program behavior following the forking of all child processes. At this point most program blocks have been copied into main memory and little access is made to the backup device.

The backup memory performs the operations described in Section 2.3, and uses a single delay parameter to model the time required to handle each incoming message. For the same reasons as for the NOCACHE and CANDF main memory models, queuing effects for incoming messages are not modelled.

3.1.9 PROP Memory Debugger

This meta-model provides debugging facilities similar to those in the NOCACHE and CANDF main memory models. This module permits the distributed main memory in the PROP architecture to be examined from a central location.

3.1.10 Statistics Module

A single instance of the statistics meta-module supports the orderly gathering of statistics from all of the other model instances at regular intervals throughout a simulation. Its primary function is to sort data and write it to statistics files. This model was provided so that separate statistics files would not be needed for each instance of a CPU or memory controller.

3.1.11 Simulator Shell

The simulator shell performs the following tasks to coordinate and provide a uniform user interface to the other modules:

- Read a multiprocessor description, creating instances of CPU's, memory controllers and main memory models, and connecting them.

- Initialize the model instances.
- Furnish an interactive shell for accessing the interactive debugging commands available from particular model instances.

3.2 Operating System Support

As part of the simulation process it was necessary to provide partial operating system support in the form of a kernel and a set of utility functions to perform such things as input, output and interprocess synchronization. The stub of a UNIX-like operating system was written to minimize the difficulty of simulating existing parallel programs written for the DYNIX operating system on the Sequent multiprocessor. The stub operating system is composed of three parts:

- a simple kernel;
- a set of standard library functions;
- a set multiprocessing library functions;

Support was limited to applications written in the C programming language.

A simple kernel was required to start up on free processors the child processes spawned by a parent. At the beginning of a simulation of an N processor system, a single processor begins executing the benchmark program (the parent) while the $N-1$ free processors execute the kernel. The kernel begins by examining a globally shared queue of processes ready to be started. If this queue is empty, the kernel indicates that its respective processor is available for work on another shared queue holding a list of free processors. Whenever a process is forked by the parent process, it is assigned to a free processor if one is available, or an entry is made in the ready queue. Processors are suspended while the wait for processes to become available on the ready queue.

The standard library consists of commonly used functions such as *printf, scanf* and *strcpy*. Each function was implemented in one of two ways: as compiled simulator code or as C code which is assembled and linked with the benchmark assembly code in Step 2 of the simulation sequence at the beginning of this chapter. Table 3.1 lists the provided functions and how they were implemented. Although the functions implemented as simulator code

Function	Implementation
printf, sprintf, fprintf	compiled
scanf, sscanf, fscanf, fgets	compiled
fopen, fclose, fflush	compiled
strcpy, strcat, strcmp, strlen	assembled
malloc, calloc	assembled
atof, atoi	compiled
sqrt, log, exp, abs	compiled
random	compiled
times	compiled
getrusage	compiled
abort	compiled
exit	compiled

Table 3.1: Standard Library Functions

do not accurately model operating system references, they were used relatively infrequently in the computationally intensive portions of the benchmarks.

The multiprocessing library provides a set of functions used to invoke child processes, allocate shared memory and manipulate synchronization variables. Subsets of the routines provided by the *DYNIX* multitasking and multiprogramming libraries [Seq86] (Table 3.2) were implemented, with all of the functions implemented as C code to be assembled and linked with benchmark programs. *s_lock* and *s_unlock* were implemented as a simple spin lock built upon test-and-set and clear SYNCOP's (Figure 3.5). A more efficient imple-

```
s_lock(lock l) {
    while(test-and-set(l));
}

s_unlock(lock l) {
    clear(l);
}
```

Figure 3.5: Simple Implementation of a Lock

mentation of *s_lock* and *s_unlock* in a multiprocessors with caches would provide a *shadow* variable for each lock so that a "spinning" processor performs most of its spinning on a copy of the shadow in its cache, thus reducing network traffic (Figure 3.6). In this implementa-

Function	Implementation
fork	assembled
shmalloc	assembled
shsbrk	assembled
s_init_barrier	assembled
s_wait_barrier	assembled
s_init_lock	assembled
s_lock	assembled
s_unlock	assembled
m_get_myid	assembled
m_set_procs	assembled
m_fork	assembled
m_sync	assembled
m_next	assembled

Table 3.2: Multiprocessing Library Functions

```
s_lock(lock l) {
    while (TRUE) {
        if (l.shadow == 0) {
            if (test-and-set(l.lock) == 0) {
                l.shadow = 1;
                return;
            }
        }
    }
}

s_unlock(lock l) {
    clear(l.lock);
    l.shadow = 0;
}
```

Figure 3.6: Better Implementation of a Lock

tion each lock is a structure with two components: *lock*, the synchronization variable which is uncacheable, and *shadow*, the shadow-variable which is cacheable and on which most of the spinning takes place. Although this implementation provides better performance when there is a great deal of contention for a lock, it greatly increases simulation time because much more spinning takes place on a shadow because of the ten-fold reduction in access time for cached data. For this reason, and also because the contention for locks in the benchmarks was not excessive, the simpler implementation of a lock was used.

The way of implementing *s_wait_barrier* was similarly chosen to improve simulation efficiency. Instead of building the barrier functions using the lock mechanism described above, the implementation of Figure 3.7 was used. Here a barrier structure is composed of:

- *lock*: a lock to prevent more than one process from modifying the structure at a time;
- *cnt*: a counter to indicate the number of processors that have reached the barrier;
- *n*: the number of processors which will check in at the barrier;
- *id_list*: a list of the identifiers of the processors checked in at the barrier;

The implementations of "suspend" and "restart" simply stop and restart the affected processor without performing any context switch; for the purposes of simulation the possibility of interrupts or multiple parallel programs sharing a processor is neglected.

3.3 Benchmarks

The benchmarks are:

1. *simpl2*, a trivial parallel program in which a specified number of children are forked, each of which iteratively prints a simple message and waits at a barrier before repeating.

2. *ssim*, a parallel simulation of a simple stochastic model of a multiprocessor. The program executes the statically defined schedule of a synchronous data flow [LM87] representation of a simple multiprocessor.

3. *genie*, a parallel topological array compactor useful in the layout of VLSI circuits. It uses an algorithm based on simulated annealling–a probabilistic optimization technique [DN87].

```
s_init_barrier(barrier b, int n) {
    s_init_lock(b.lock);
    b.n = n;
    b.cnt = 0;
}

s_wait_barrier(barrier b) {
    s_lock(b.lock);
    b.cnt = b.cnt + 1;
    if (b.cnt != b.n) {
        /* myid is the identifier of the calling processor */
        b.id_list[b.cnt] = myid;
        s_unlock(b.lock);
        suspend;
    }
    else {
        b.cnt = 0;
        for (i = 0; i < b.n - 1; i++) {
            restart(b.id_list[i]);
        }
    }
    s_unlock(b.lock);
}
```

Figure 3.7: Implementation of a Barrier

Benchmark	Lines	Memory			No. of Refs.	
		code	local	shared	parent	child
simpl	180	9k	68	1040	170k	52.4k
ssim	470	15k	200k	750	280k	90k
genie	2300	46k	12k	1.5M	9M	523k
verf	2600	47k	68k	460k	2M	760k

Table 3.3: Benchmark Characteristics

4. *verf*, a parallel verifier of combinational Boolean logic [M*87].

Some characteristics of the benchmarks, including program size, memory usage and reference behavour are shown in Table 3.3. These benchmarks were all originally written and debugged on the Sequent shared memory multiprocessor. Benchmarks 2 to 4 are computationally intensive and require little operating system intervention throughout their execution.

3.4 Simulated Architectures

Table 3.4 shows the default parameters chosen for the three multiprocessor architectures for which simulations were performed. Unless specified otherwise, these parameters were used for all of the simulation results in the next chapter. It was assumed that the time to cross the network was 10 times the delay to service a cache hit at a memory controller. Furthermore, the time to fetch a single word from main memory was assumed to be the same as the time to fetch an entire 4 word sub-block; such an assumption applies to a network with high latency and high bandwidth. The value assigned to the PROP controller delay was made under the assumption that a smaller, faster cache exists between the main memory cache and the CPU. The size of the PROP controller cache was assumed to be 10 times the size of the CANDF and NOCACHE caches. This assumed that the PROP two-level hierarchy composed of a 100kByte first level cache followed by a 1MByte block of distributed memory is equivalent to a 1MByte first level cache. As justification, if the top level cache as a hit ratio of 0.97, and the second level latency is 10 times the first, the latency of the two levels is $0.97(1) + 0.03(10) = 1.27$, which is only 27% longer than the minimum.

MODEL[a]	PARAMETER	VALUE
CPU	Delay	100
NOCACHE Controller	Delay	100
	Cache Size	100kB
	Block Size	16B
NOCACHE Main Memory	Delay	100
	Size	1MB
CANDF Controller	Delay	100
	Cache Size	100kB
	Block Size	512B
	Sub-Block Size	16B
CANDF Main Memory	Delay	100
	Size	1MB
PROP Controller	Delay	100
	Cache Size	1MB
	Block Size	512B
	Sub-Block Size	16B
PROP Backup Memory	Delay	100
	Size	10MB

[a]All models assume a network delay of 1000.

Table 3.4: Default Parameters of Simulated Architectures

Chapter 4

Simulation Results

Table 4.1 shows the cpu reference characteristics and cache hit ratios for simulations using the default parameters of Section 3. Since different coherence schemes result in different dynamic referencing behavior at synchronization points, the breakdown of references for a given benchmark varies slightly. In general, the use of smaller blocks results in improved hit ratios, which corresponds with the sector cache study of [HS84]. As expected, the PROP scheme with its larger effective cache size results in fewer misses in the larger benchmarks, *genie* and *verf*.

Figures 4.1 and 4.2 show the average access time, f_a, and average network traffic, f_n, of the CANDF and PROP schemes normalized with respect to the NOCACHE scheme. $f_a = 0.5$ means that the average access time of a particular scheme is half that of the NOCACHE scheme; similarly, $f_n = 0.5$ means that the average network traffic of a particular scheme is half that of the NOCACHE scheme. The per-process access time for each caching scheme was calculated by dividing the run-time of the process by the total number of cpu reads, writes and SYNCOPs to shared and unshared data. Time spent sleeping at a barrier was subtracted from the total run-time. Per process average network traffic was calculated as the total number of normalized network transactions divided by the total number of cpu reads, writes and SYNCOPs. As mentioned previously, one normalized network transaction corresponds to the transfer of 4 bytes or less across the network (one way only). The numbers reported here are averages of the per-process average access time and network traffic for all child processes. The results of Figures 4.1 and 4.2, and all other results presented in this section, were derived from the portions of the simulations immediately following the forking of all child processes.

Benchmark and Coherence Scheme		f_s [a]	f_{ns} [b]	f_{sy} [c]	f_{sw} [d]	128B Block		512B Block		2kB Block	
						h_s [e]	h_n [f]	h_s	h_n	h_s	h_n
simpl2	n	0.27	0.64	0.09	0.34	-	0.997	-	0.997	-	0.997
	c	0.24	0.59	0.17	0.34	0.992	0.997	0.992	0.997	0.992	0.997
	p	0.26	0.64	0.10	0.34	0.991	0.997	0.991	0.997	0.991	0.997
ssim	n	0.30	0.68	0.02	0.28	-	0.995	-	0.995	-	0.995
	c	0.29	0.66	0.05	0.28	0.998	0.996	0.998	0.996	0.998	0.996
	p	0.30	0.68	0.02	0.28	0.998	0.995	0.998	0.995	0.998	0.995
genie	n	0.39	0.59	0.02	0.18	-	0.999	-	0.999	-	0.999
	c	0.38	0.57	0.05	0.19	0.972	0.999	0.964	0.998	0.943	0.990
	p	0.39	0.59	0.02	0.18	0.973	0.999	0.974	0.999	0.974	0.999
verf	n	0.37	0.62	0.01	0.23	-	0.999	-	0.999	-	0.999
	c	0.37	0.61	0.02	0.22	0.937	0.996	0.916	0.992	0.898	0.988
	p	0.37	0.62	0.01	0.22	0.951	0.998	0.953	0.998	0.956	0.999

[a] f_s is the fraction of references to shared writeable data.
[b] f_{ns} is the fraction of references to non-shared writeable data.
[c] f_{sy} is the fraction of references to synchronization variables.
[d] f_{sw} is the fraction of references to shared writeable data which are writes.
[e] h_s is the hit ratio for references to shared writeable data.
[f] h_n is the hit ratio for references to non-shared writeable data.

Table 4.1: Simulation Statistics

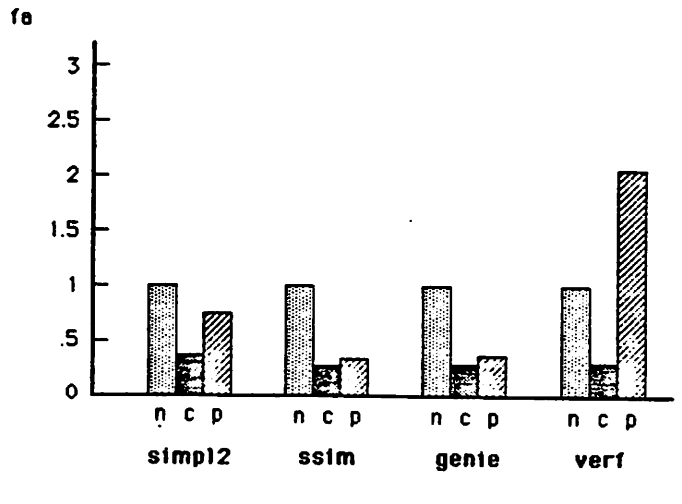

Figure 4.1: Normalized Access Time with Default Parameters

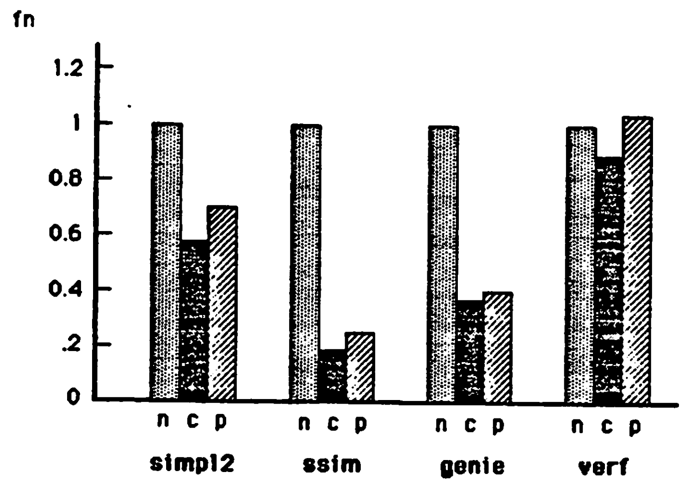

Figure 4.2: Normalized Network Traffic with Default Parameters

For all benchmarks the CANDF scheme provides substantial speedup over NOCACHE, and for all but *verf* provides a substantial reduction in network traffic. The PROP scheme provides substantial improvements in f_a and f_n over NOCACHE, with values of f_a and f_n slightly larger than those for CANDF. For the *verf* benchmark, PROP performs worse than NOCACHE in terms of f_a and f_n. This is in spite of the fact that Table 4.1 indicates that the PROP scheme exhibits fewer misses for the verf benchmark. This suggests that queuing effects are the cause of the poor performance.

Figures 4.3 and 4.4 show the absolute values of average access time and network traffic for the benchmark set. Neglecting the *verf* benchmark, the average access time of the *candf* and *prop* schemes varies from about 3 to 8 times the minimum cache processing time of 100.

To investigate the cause of CANDF's superior values of f_a relative to PROP, simulations were performed in which the queuing effects at the network inputs to PROP memory controllers were ignored. The results (Figure 4.5) show that these queuing effects account for the reduced performance of the PROP scheme, and also the particularly poor performance of PROP on the *verf* benchmark. The severe impact of the queuing effects can be explained by considering that when the parent process forks off the child processes all of the shared data on which they operate resides in the memories of only two processors. The queuing effects are therefore extreme because the majority of the network transactions of the many child processes are directed to only a few processors; this phenomena does not

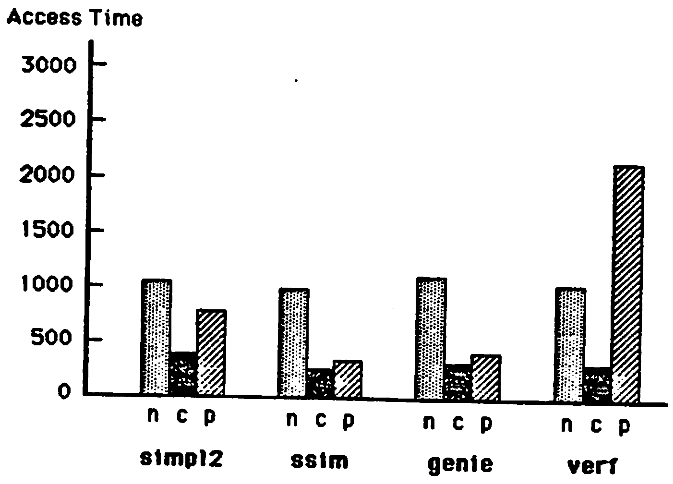

Figure 4.3: Absolute Access Time with Default Parameters

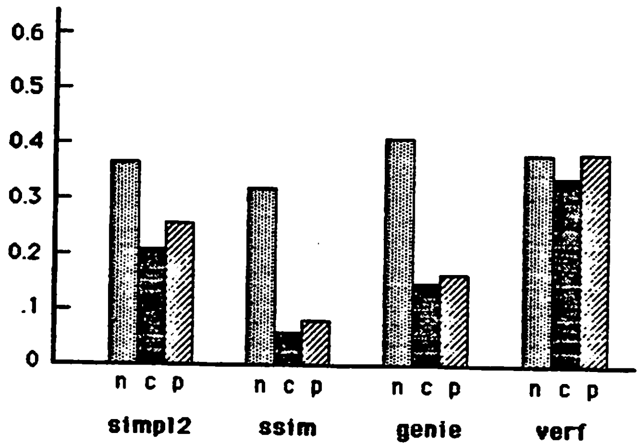

Figure 4.4: Absolute Network Traffic with Default Parameters

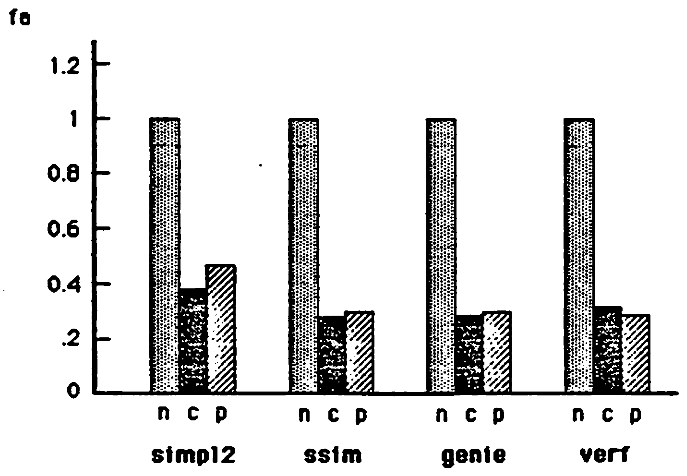

Figure 4.5: Normalized Access Time, Neglecting PROP Queuing Effects

arise in the NOCACHE and CANDF schemes because of relatively uniform interleaving at their main memories. The results of Figure 4.5 indicate that for the PROP scheme to be viable some technique must be applied to spread shared memory across the processors in a uniform way. One suitable technique may be to have the controllers displace blocks that receive many more accesses from the network than from the local CPU. Another viable solution may be to limit the number of copies of a block to some value much less than the total number of caches in the system.

Figures 4.6 to 4.7 present the results of additional simulation results which investigate:

- The effect of block size on f_a (Figures 4.6 and 4.7).
- The effect of the network model on f_a (Figures 4.8 to 4.11).
- The average number of copies of a shared block (Figure 4.12).
- The average number of valid sub-blocks with a block (Figure 4.13).
- The average number of invalidations required per shared write (Figure 4.14).
- The amount of parallelism in each benchmark (Figure 4.15).

As Figures 4.6 and 4.7 show, f_a improved for both the CANDF and PROP schemes slightly as block size is decreased, except for the *verf* benchmark, in which the improvement is about 65% for the PROP case. For each simulation the sub-block size was kept at 16

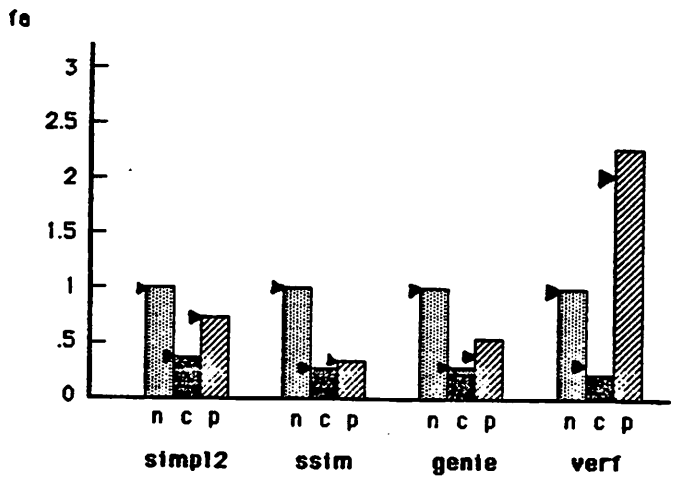

Figure 4.6: Normalized Access Time with 2kB Block Size (Markers denote results for 512B block)

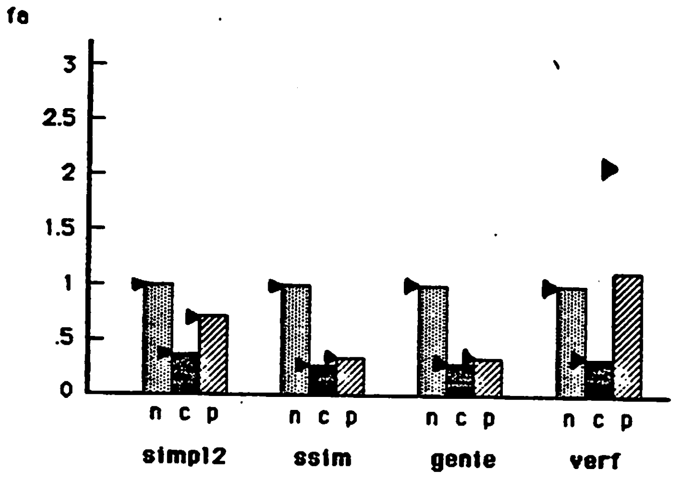

Figure 4.7: Normalized Access Time with 128B Block Size (Markers denote results for 512B block)

bytes, and queuing effects were taken into account in the PROP controller model. Smaller block sizes result in better performance because fewer sub-blocks need to be written back on a displacement, and there is less *false sharing*. False sharing occurs when the block containing a piece of data has copies in one or more caches in which the particular data will never be accessed. This arises when different pieces of data within the same block are referenced in different caches—although each datum is unshared, the fact that the containing block is in two caches makes it *appear* that they are shared. Consequently, redundant invalidations are occasionally sent, resulting in poorer performance. The results of these simulations show, however, that in spite of the preceding effects the CANDF and PROP schemes do not require small block sizes to obtain reasonable performance.

The effects of different network characteristics are presented in Figures 4.8 to 4.11. Figure 4.8 shows the values of f_a obtained with a uniform network 10 times slower than

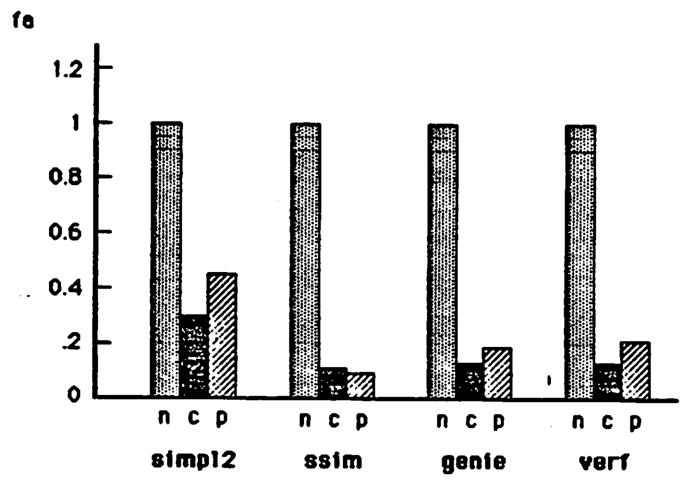

Figure 4.8: Normalized Access Time with 100:1 Uniform Network

the default network: here the time to cross the network is assumed to be 100 times the delay to service a cache hit. Although the average value of f_a improves by about a factor of 3, absolute performance is still seriously affected by the much slower network (Figure 4.9). Figure 4.10 shows the values of f_a obtained with a cube network in which the delay between adjacent processor is the same as the uniform delay in the default uniform network. Compared to the results for the fast, uniform network, the average value of f_a is improved by about a factor of 2, and absolute performance of the CANDF and PROP schemes degrades by about 40 % (Figure 4.11).

Figure 4.12 presents the average number of copies of a shared block observed in simulations of the CANDF and PROP schemes. As expected, the much larger effective cache size of the PROP scheme results in a larger average number of copies—this is because copies are much less likely to be displaced in a much larger cache. Figure 4.12 reveals that

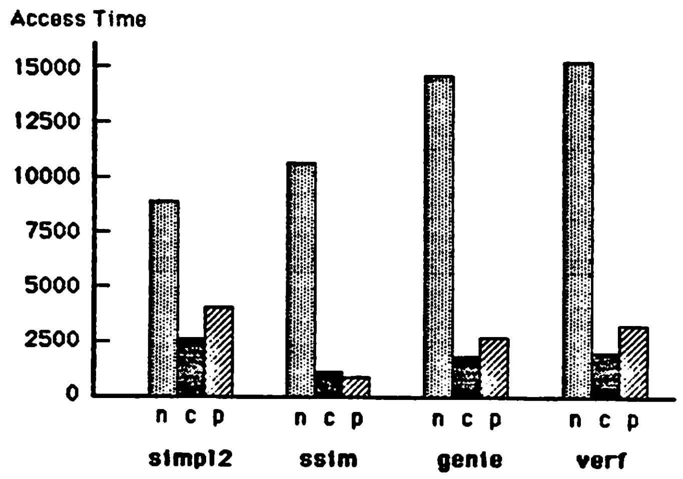

Figure 4.9: Absolute Access Time with 100:1 Uniform Network

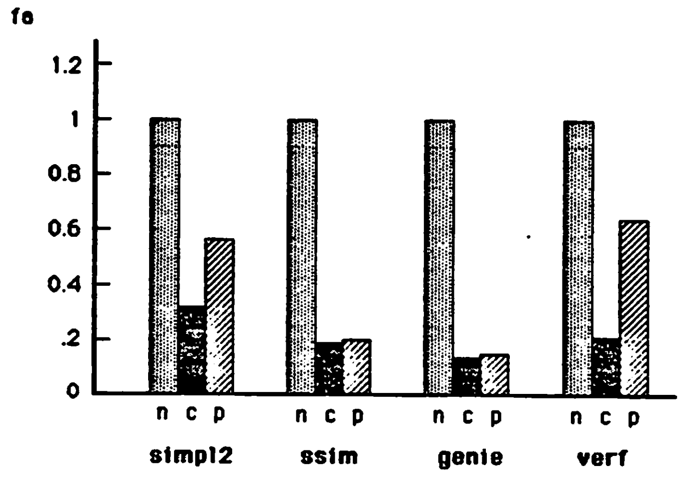

Figure 4.10: Normalized Access Time with Cube Network

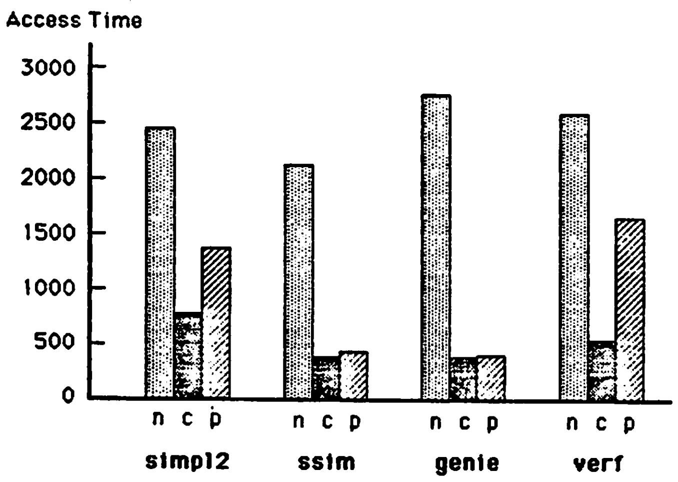

Figure 4.11: Absolute Access Time with Cube Network

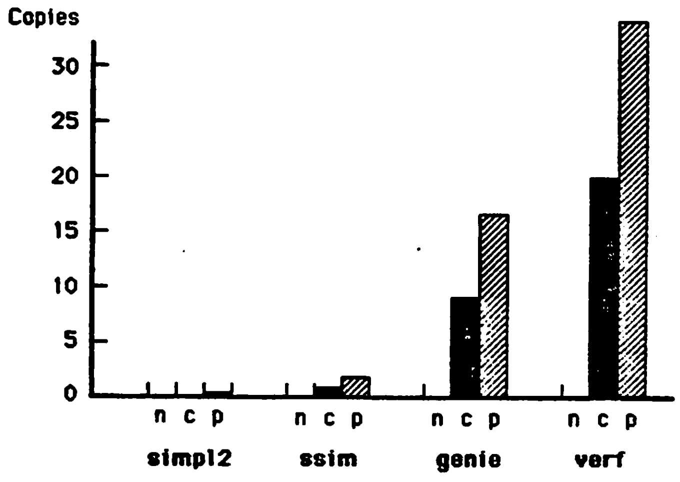

Figure 4.12: Average Number of Copies of a Shared Block

most shared blocks reside in a large fraction of the 64 caches, suggesting that more than a few cache identifiers are required in the Censier and Feautrier scheme.

The average fraction of a block that is occupied by valid data (Figure 4.13) shows that only a minor portion of a block is in use at any particular time. This suggests that

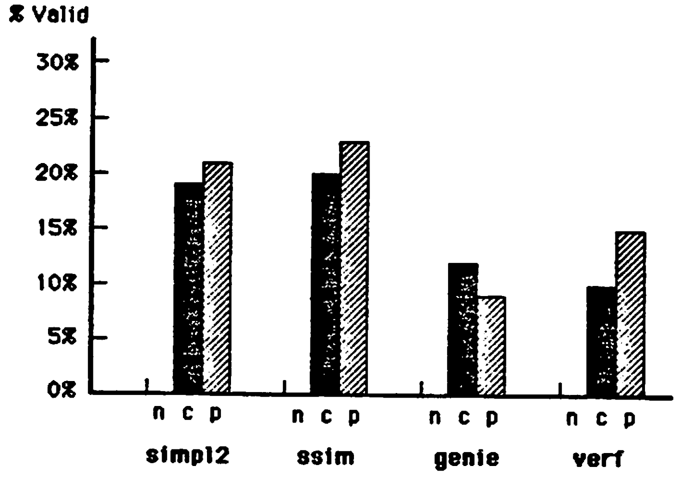

Figure 4.13: Percentage of Valid Sub-Blocks within a Block

smaller block sizes result in a larger fraction of memory holding valid data, and correspondingly higher hit ratios. Table 4.1 verifies that the hit ratio improves as block size is decreased. In spite of the higher hit ratios obtainable with small block sizes, the results in Figures 4.6 and 4.7 suggest that the delay from references to synchronization variables (which always bypass the cache) and waiting at barriers dominates the delay from cache misses. The benefit of small block sizes is thus minimal.

Figure 4.14 shows that, on average, sub-block invalidations are issued to over half of the available processors for each write to shared memory. This may be due to false sharing caused by large cache sizes (100kB and greater) and the large block size. As expected, the larger cache size in the PROP scheme results in more invalidations per shared write than in the CANDF scheme. It is somewhat surprising that in spite of the large number of invalidations per shared write, the CANDF and PROP schemes provide better performance and reduced network traffic than the NOCACHE scheme. Since many of these invalidations may be due to old blocks which have not been displaced and will not be referenced for a long time to come, it may be beneficial to modify the controllers so that blocks which are referenced substantially more frequently by the network than by the local CPU get

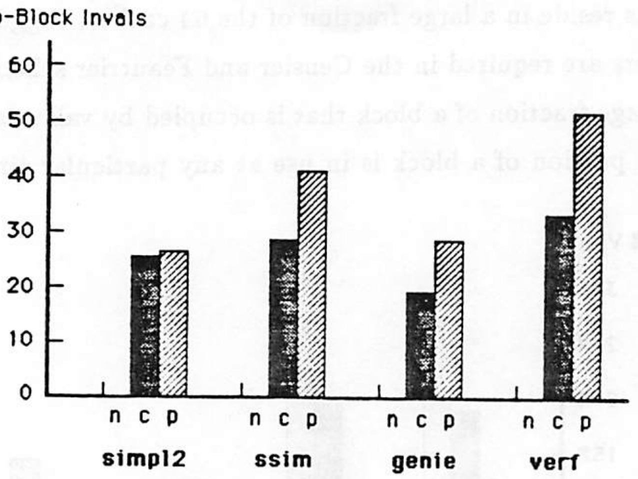

Figure 4.14: Average Number of Invalidations Issued per Shared Write

displaced. Alternatively, limiting the number of copies of a block may also be an effective solution. These same modifications were also suggested as ways to alleviate the adverse queuing effects in the PROP controller.

The final set of simulation results indicate the amount of parallelism in benchmarks 2 to 4 by comparing the execution times obtained for 32 and 64 simulated processors. Figure 4.15 shows the speedups from 32 to 64 processors, defined as the ratio of the execution times for the two cases. Substantial speedups of about 50% are obtained for the NOCACHE

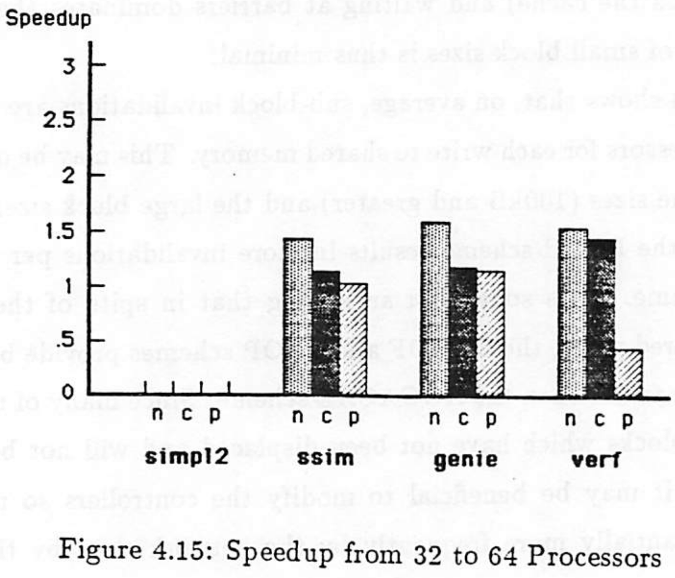

Figure 4.15: Speedup from 32 to 64 Processors

architecture, and more modest gains of 18% or less for CANDF and PROP. Since the programs executed 2 or more times slower with the NOCACHE scheme, the granularity of parallel tasks is effectively reduced, providing one reason why NOCACHE exhibits greater speedups from 32 to 64 processors. In other words, the time taken to perform computation between synchronization points is reduced in the CANDF and PROP schemes, but the time spent serializing at synchronization points is not reduced because accesses to synchronization variables bypass the cache. When the fraction of computation time spent in purely serial activity increases, the maximum speedup attainable using multiple processors decreases (by Amdahl's "law" [Qui87]). For the PROP architecture, the *verf* benchmark executes faster on 32 processors than on 64. This anomaly is probably caused by the queuing effects at the PROP controller discussed earlier: since the memory of the parent processor owns the majority of shared data blocks, the contention of the other processors trying to get copies from the parent memory increases with the number of processors.

Chapter 5

Conclusions

This section summarizes the major results of this work. Three hardware coherence schemes for large shared memory multiprocessors were compared using instruction-level simulations of four benchmark programs. The first coherence scheme was the simplest possible, in which no shared writeable data is cached. The second was a sectored version of Censier and Feautrier's directory method that reduces the main memory tag size to an acceptable value. The third was a modified version of the second in which main memory is distributed among processors and used as a second level cache.

As expected, caching shared writeable data is effective at substantially reducing average memory access time and average network traffic for real multiprocessor applications in a shared memory multiprocessor. The simulations showed that for four benchmark programs both access time and network traffic could be consistently reduced by a factor of 2-3 using the sectored version of Censier and Feautrier's directory method. In addition, it was shown that sectoring Censier and Feautrier's scheme reduces tag overhead to less than 15 % for a sub-block size of 16 bytes, page size of 1024 bytes or larger, and 512 or fewer processors.

A simulation study of the second variation of Censier and Feautrier's coherence scheme showed that although a reduction in misses was made possible by using main memory as a second level cache, performance did not improve beyond that of the sectored scheme. In fact, it was worse because the average access time of the third scheme is severely affected by adverse queuing effects at the memory controllers. With the queuing effects eliminated, performance was comparable to that obtained with the sectored Censier and Feautrier scheme. For the benchmarks considered, the performance of the third coherence scheme does not justify its considerable added complexity.

Results for the sectored version of Censier and Feautrier's scheme showed that for a given sub-block size, increasing the block size increased the number of misses. The increase in miss ratio, however, did not appreciably hurt performance for large block sizes. Delays due to misses appeared to be dominated by delays due to references to synchronization variables; such references always bypass the cache, and their frequency of occurrence is only weakly affected by block size.

The sectored Censier and Feautrier scheme exhibited an average number of invalidations per shared write approaching half the number of processors. Since other studies have suggested that data is typically only shared by a small number of processors at a time [A*88], it appears that sectoring greatly intensifies false sharing. A way to reduce the number of invalidations may be to displace cache blocks that receive many references from the network and few references from the local processor. This issue deserves further study.

There remain many ways in which this study can be expanded. First, more benchmarks, especially those with greater parallelism, should be investigated. Simulations using more processors should also be performed to see if the conclusions of this study scale. Simulation accuracy should be verified using more accurate network models that take into account some of the queuing effects that were ignored. A comparison of the coherence schemes considered here and several of the software methods is needed to determine if complex hardware support is really necessary. Analytic models should be developed and compared with these simulation results to refine current, and guide future, shared memory multiprocessor designs. Lastly, and ultimately, some multiprocessor hardware must be implemented to reveal the subtle and not-so-subtle effects that are frequently not revealed by simulation.

Bibliography

[A*84] T. Axelrod et al. A simulator for MIMD performance prediction: application to the S-1 MkIIa multiprocessor. *Parallel Computing*, 1:237–274, 1984.

[A*88] A. Agarwal et al. An evaluation of directory schemes for cache coherence. In *Proceedings of the International Symposium on Computer Architecture*, pages 280–289, May 1988.

[AB84] J. Archibald and J-L. Baer. An economical solution to the cache coherence problem. In *Proceedings of the International Symposium on Computer Architecture*, pages 355–362, 1984.

[AB86] J. Archibald and J. Baer. An evaluation of cache coherence solutions in shared-bus multiprocessors. *ACM Transactions on Computer Systems*, 4(4):273–298, Nov. 1986.

[AS83] G. R. Andrews and F. B. Schneider. Concepts and notations for concurrent programming. *Computing Surveys*, 15(1):3–43, March 1983.

[BC86] Beranek Bolt and Newman Computer Corp. The butterfly multiprocessor: overview. 1986.

[BD86] P. Bitar and A. M. Despain. Multiprocessor cache synchronization: issues, innovations, evolution. In *Proceedings of the International Symposium on Computer Architecture*, pages 424–433, June 1986.

[CF78] L. M. Censier and P. Feautrier. A new solution to coherence problems in multicache systems. *IEEE Transactions on Computers*, C-27(12):1112–1118, Dec. 1978.

[CV88] H. Cheong and A. V. Veidenbaum. A cache coherence scheme with fast selective invalidation. In *Proceedings of the International Symposium on Computer Architecture*, pages 299–307, May 1988.

[Den70] P. J. Denning. Virtual memory. *Computing Surveys*, 2(3):153–189, Sept. 1970.

[DN87] S. Devadas and A. R. Newton. Topological optimization of multiple level array logic. *IEEE Transactions on Computer-Aided Design*, Nov. 1987.

[EK88] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the International Symposium on Computer Architecture*, pages 373–383, June 1988.

[G*83a] D. Gajski et al. Cedar–a large scale multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, pages 514–529, August 1983.

[G*83b] A. Gottlieb et al. The NYU Ultracomputer–designing an MIMD shared memory parallel computer. *IEEE Transactions on Computers*, C-32(2):175–189, Feb. 1983.

[Goo83] J. R. Goodman. Using cache memory to reduce processor-memory traffic. In *Proceedings of the International Symposium on Computer Architecture*, pages 124–131, June 1983.

[GW88] J. R. Goodman and P. J. Woest. The Wisconsin Multicube: a new large-scale cache-coherent multiprocessor. In *Proceedings of the International Symposium on Computer Architecture*, pages 422–433, May 1988.

[HS84] M. D. Hill and A. J. Smith. Experimental evaluation of on-chip microprocessor cache memories. In *Proceedings of the International Symposium on Computer Architecture*, pages 158–166, June 1984.

[K*85] R. H. Katz et al. Implementing a cache consistency protocol. In *Proceedings of the International Symposium on Computer Architecture*, pages 276–283, June 1985.

[LM87] E. A. Lee and D. G. Messerschmitt. Static scheduling of synchronous data flow programs for digital signal processing. *IEEE Transactions on Computers*, C-36(1):24–35, Jan. 1987.

[M*87] H. T. Ma et al. Logic verification algorithms and their parallel implementation. In *Proceedings of the Design Automation Conference*, pages 283–290, July 1987.

[McC84] E. M. McCreight. The Dragon computer system. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, 1984.

[P*85] G. F. Pfister et al. The IBM research parallel processor prototype (RP3): introduction and architecture. In *Proceedings of the International Symposium on Computer Architecture*, June 1985.

[PN85] G. F. Pfister and V. A. Norton. Hot spot contention and combining in multistage interconnection networks. *IEEE Transactions on Computers*, C-34(10):943–948, Oct. 1985.

[PP84] M. S. Papamarcos and J. H. Patel. A low-overhead coherence solution for multiprocessors with private cache memories. In *Proceedings of the International Symposium on Computer Architecture*, pages 348–355, Jan. 1984.

[Qui87] M. J. Quinn. *Designing Efficient Algorithms for Parallel Computers.* McGraw-Hill, 1987.

[Ros85] C. D. Rose. Encore eyes multiprocessor market. *Electronics*, :118–119, July 8 1985.

[RS84] L. Rudolph and Z. Segall. Dynamic decentralized cache schemes for mimd parallel processors. In *Proceedings of the International Symposium on Computer Architecture*, pages 340–347, June 1984.

[Seq84] Sequent Computer Systems, Inc. Balance 8000 technical summary. Nov. 1984.

[Seq86] Sequent Computer Systems, Inc. Balance guide to parallel programming. 1986.

[Smi82] A. J. Smith. Cache memories. *Computing Surveys*, 14(3):473–530, Sept. 1982.

[Smi85] A. J. Smith. CPU cache consistency with software support and using one time identifiers. In *Proceedings of the Pacific Computer Communications Conference*, pages 153–161, 1985.

[SS86] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the International Symposium on Computer Architecture*, pages 414–423, 1986.

[Sto87] H. S. Stone. *High-Performance Computer Architecture*. Addison-Wesley Publishing Company, 1987.

[TS87] C. P. Thacker and L. C. Stewart. Firefly: a multiprocessor workstation. In *Proceedings of the International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 164–172, Oct. 1987.

[Ulr78] E. G. Ulrich. Event manipulation for discrete simulations requiring large numbers of events. *Communications of the ACM*, 21(9):777–785, Sept. 1978.

[Wil87] A. W. Wilson Jr. Hierarchical cache/bus architecture for shared memory multiprocessors. In *Proceedings of the International Symposium on Computer Architecture*, pages 244–252, June 1987.

[Y*85] W. C. Yen et al. Data coherence problem in a multicache system. *IEEE Transactions on Computers*, C-34(1):56–65, Jan. 1985.