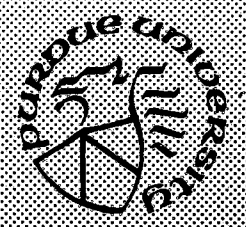
5-1-1990

Directory Based Cache Coherency Protocols for Shared Memory Multiprocessors

Craig Warner
Purdue University

FILE COPY



Directory Based Cache Coherency Protocols for Shared Memory Multiprocessors

Craig Warner

TR-EE 90-33
May 1990

School of Electrical Engineering
Purdue University
West Lafayette, Indiana 47907

DIRECTORY BASED CACHE COHERENCY PROTOCOLS FOR SHARED MEMORY MULTIPROCESSORS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Craig Warner

In Partial Fulfillment of the

Requirements for the Degree

of

Masters of Science in Electrical Engineering

May 1990

To my Father and Mother who have cared so deeply

ACKNOWLEDGEMENTS

I would like to use this time and space to thank Professor D. Meyer for listening to my fully cooked as well as many of my half-baked ideas. I would also like to thank the rest of my committee: Professor R. Fujii and Professor H. Dietz. For encouraging me to go to graduate school, I thank Professor S. Kothari.

TABLE OF CONTENTS

Page

LIST OF FIGURES

Figure Page

LIST OF ABBREVIATIONS

System parameters

B	-	Size of a cache block
D	-	The set of processors which the directory thinks has a particular block cached
|D|	-	The number of processors which receive invalidation messages when a block is invalidated
G	-	Number of Clusters in a system
GS	-	Active global table size
h	-	Average hit ratio of the private caches
K	-	Associativity of the private caches
M	-	Number of cache blocks in main memory
MM	-	Number of memory modules (usually N)
N	-	Number of processors
Ncb	-	Number of cache blocks in each private cache
Ns	-	Number of sets in each private cache
Nsm	-	Number of memory submodules
P	-	The set of processor which have a particular block cached (the present set)
|P|	-	The number of processor which have a particular block cached

Program parameters

Fw	-	Fraction of references which are writes
Nsb	-	Number of shared cache blocks
qro	-	Fraction of references which are to read only data
qs	-	Fraction of references which are to shared data

Other

RO	-	Read Only
RW	-	Read Write

ABSTRACT

Warner, Craig. M.S.E.E., Purdue University, May 1990. Directory Based Cache Coherency Protocols For Shared Memory Multiprocessors. Major Professor: David Meyer

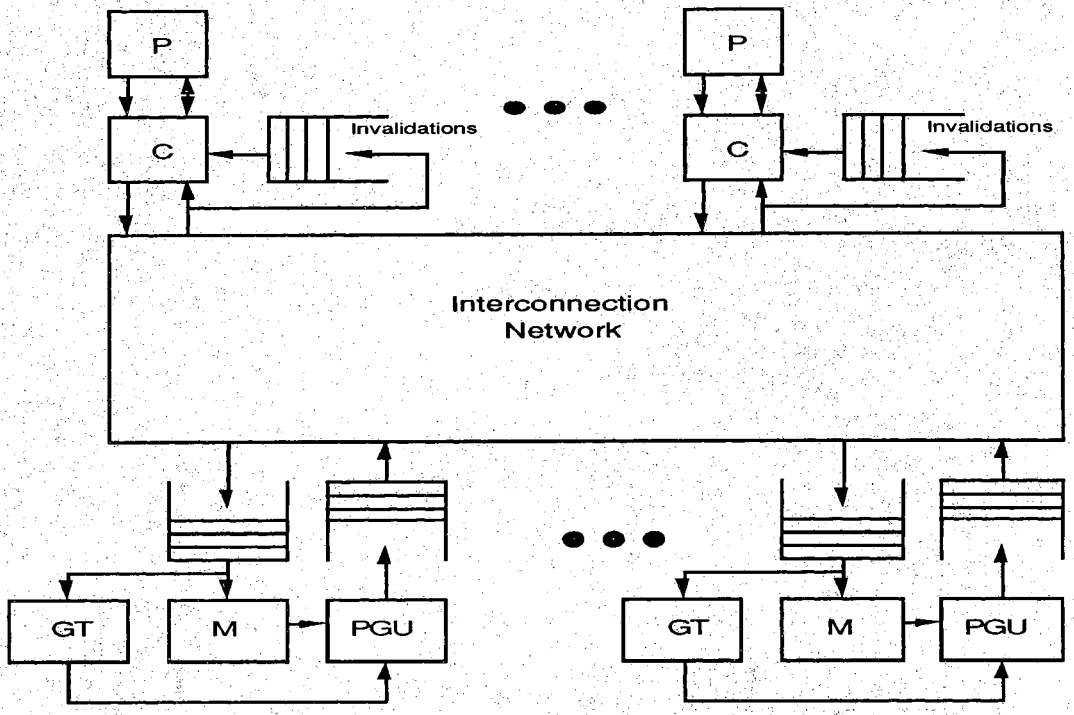
Directory based cache coherency protocols can be used to build large scale, weakly ordered, shared memory multiprocessors. The salient feature of these protocols is that they are interconnection network independent, making them more scaleable than snoopy bus protocols. The major criticisms of previously defined directory protocols point to the size of memory needed to store the directory and the amount of communication across the interconnection network required to maintain coherence. This thesis tries solving these problems by changing the entry format of the global table, altering the architecture of the global table, and developing new protocols. Some alternative directory entry formats are described, including a special entry format for implementing queueing semaphores. Evaluation of the various entry formats is done with probabilistic models of shared cache blocks and software simulation. A variable length global table organization is presented which can be used to reduce the size of the global table, regardless of the entry format. Its performance is analyzed using software simulation. A protocol which maintains a linked list of processors which have a particular block cached is presented. Several variations of this protocol induce less interconnection network traffic than traditional protocols.

1. PRELIMINARIES

1.1 General Architecture

Directory based cache coherency protocols are a way of making the memory in a multiprocessor system logically the same for all processors. The shared memory paradigm is desirable from the programmer's perspective because of its conceptual simplicity. All processor-to-processor communication can be performed through accessing shared memory locations. Because VonNeumann architectures are limited by memory bandwidth, the shared memory must also be fast. In large multiprocessor systems, there is great disparity between the main memory bandwidth and rate at which processors generate memory references. Thus, the need for some way to satisfy the majority of memory references without using main memory. This has lead many researchers to consider systems with large private caches. These caches must be on the processor side of the interconnection network to be effective. With the inclusion of these caches comes the coherency problem of trying to maintain the same, "up to date" data from the point of view of all the processors. In directory based cache coherency protocols, a global table records the current state of the cache blocks (lines) in the system. The directory, or global table, is distributed across the memory modules, and is used for every reference to main memory. The global table stores which processor's private caches currently have the block cached, and whether the block is inconsistent with memory. The set of processor which have a block cache is called the **present set** (P).

The general processor and memory organization is shown in Figure 1.1.



P - Processor C - Private Cache GT - Global Table Module M - Memory Module
PGU - Packet Generation Unit

Figure 1.1 Processor and Memory Organization for Directory Based Protocols

The memory is divided into modules, which can be interleaved to perform block size reads and writes quickly [BrDa77]. The packet generation unit receives message requests, from the memory, to send cache blocks to processors which have had private cache read or write misses. It also receives invalidation message requests form the global table when a processor writes to a block which is cached by other processors.

1.2 Why Directory Based Protocols?

There are many ways of implementing shared memory in multiprocessors. Some of these techniques are with software, others are with hardware, and some require both. The most promising non-directory based protocols are snoopy protocols (for bus systems) and self invalidating protocols.

The snoopy protocols are ultimately limited by the rate at which addresses may be placed on the address bus. Unfortunately, the throughput of a bus is roughly inversely proportional to the number of processors placed on the bus. Simulation results [ArBa85] [YaBL89] show that the performance of these protocols levels off at around 32 processors. Sequent Computer [Sequ87] sells multiprocessor systems, using snoopy bus protocols, with as many as 20 processors, but no more. Furthermore, RISC processors have higher memory throughput requirements, which further complicates the bus bandwidth problem.

Self invalidating protocols [ChVe88][MiBe89] involve compile analysis of programs to determine, for each reference, whether or not the reference should be serviced by the cache or main memory. As said by Min and Baer [MiBa89], "It is clear that directory based protocols ... will always have higher hit ratios than self-invalidating schemes. On the other hand, there will be less network traffic in the self-invalidating schemes.."

Because of the network independence of directory based schemes, non-bus interconnection networks can be used to satisfy the memory bandwidth required by high speed CPUs or many moderate speed CPUs, while maintaining higher cache hit ratios than self-invalidating protocols yield.

1.3 Operating System and Programming Model Assumptions

Throughout the rest of the thesis, several assumptions about the operating system, programming model, and event ordering will be made. They are:

1) All accesses to shared RW variables must be performed after accesses to synchronization variables. Synchronization variables can be semaphores when the references to shared data are in critical sections. Synchronization variables can also be directed, like the synchronization in DoAcross loops.

2) The order of execution of instructions running in parallel is not deterministic (Figure 1.2).

```
b:=3;
parbegin
   a:=b+c;
   b:=5;
parend
```

Figure 1.2 Non-deterministic Program

3) All memory accesses issued by processor p must be performed with respect to all other processors before p accesses a synchronization variable. Issued and performed as used above are precise terms. A reference is said to be issued when the reference can no longer be cancelled by the processor which initiated it. Most of the time references are issued when they enter the processor to memory interconnection network. A read from memory is said to be performed with respect to processor k when processor k can no longer initiate a write which alters the value fetched by the read. A write to memory is said to be performed with respect to processor k when processor k cannot initiate a read operation to the same location which does not receive the value stored by the write.

4) All synchronization variables are not cached.

5) Semaphores are accessed through uninterruptable read-modify-write operations. Examples of such operations are test-and-set, compare-and-swap [PeSi85], and the xmem instruction used in the Motorola 88000 [Moto88]

6) Every two shared variables which could concurrently be granted RW privilege to two different processors, must be stored in separate cache blocks.

Together the assumptions, one through six, imply Weak Ordering of events [DuSB86] [DuSB88], explained later in this chapter.

1.4 Private Caches

The private caches can perform five basic types of operations: read hits, read misses, write hits, write misses, and block ejections. How each of these five operations effects the global table is discussed below.

Read Hit

When a read hit occurs, the data is fetched from the private cache; there is no global table operation.

Write Hit

When a write hit occurs, the global table record of that block is updated; all other caches which have the block are invalidated.

Read Miss

For read misses, the global table is updated to reflect that a new processor also has the block cached. If the block is not cached or cache read only (RO), the block is fetched from the memory. If the block is in another processor's cache and is dirty, the block must be written back to memory and sent to the reading processor. The processor which has the block dirty does not need to invalidate the block.

Write Miss

When write misses occur, the global table must be updated to reflect that the processor which is performing the write is the only processor with the block and the block is dirty. If when the write miss occurs, the block is not cached or cached RO, the block is fetched from memory. If the block is cached read write (RW), then the processor which has the dirty block must send it to the writing processor. All other copies of the block must be invalidated because only one processor at a time is allowed to have a block cached RW.

Ejection

A block ejection occurs when a valid block must be removed from a cache because a new block needs to be moved into the same cache set, and there is not enough associativity to hold all the valid blocks in the set. When clean blocks are ejected, the global table may or may not be informed of the update. The effects of not updating are discussed in chapter three.

The private caches have three essential parameters, the size of the cache block (B), the number of sets in the cache (Ns), and the associativity of each set (K). The product of these three parameters equals the size of the cache, and the number of blocks in the cache (Ncb) is the number of sets multiplied by the associativity (K*Ns). The greater the associativity, the slower the access

time, but the less frequent blocks ejections are. As the cache size increases, the associativity effects on the hit rate diminish [Hill88]. This is why large caches are usually direct mapped.

For some systems the disparity between the processor speed and memory speed is so great as to warrant multilevel caches. The multiple levels of caches further complicates coherency control. One simplifying constraint which does not drastically affect performance is the **inclusion property** [BaWa88]. The inclusion property states that every block in a faster cache is also in every slower cache. This way, cache invalidation requests only need to check against the cache tags of the slowest cache. The slowest cache needs to maintain a bit for each block indicating if that block is cached by a faster level. This way, if a block is invalidated, the slowest cache knows when it must invalidate the faster cache(s).

Some RISC processors are designed to work with two private caches: a data cache and an instruction cache [Moto88]. This allows for concurrent access to instructions and data which increases parallelism. If self modifying code is not allowed, the cache coherency problem is nonexistent in the instruction cache, making it possible to use instruction caches in vary large scale multiprocessors like the BBN Monarch [RCCT90]. The ideas we discuss in this paper can be easily extended to system with separate instruction and data caches, so for simplicity we assume each processor has only one cache.

1.5 Event Ordering

In multiprocessors, there are N sequences of memory references which in a shared memory system need to be merged into one sequence. **Sequential Consistency** is merging these reference sequences in such a may that no two references from the same processor appear to execute in an order different than that specified by the program. For conceptual simplicity and definition exactness, we introduce the concept of an event order graph. In the event order graph, the references are thought of as vertices, and dependencies are thought of as directed edges. Sequential Consistency is violated if, and only if, there exits a cycle in the event ordering graph. A machine is said to maintain Sequential Consistency if, for every possible program, there does not exist an execution which violates Sequential Consistency. Formally, the event order graph is constructed in the following way.

A. Every read or write to memory corresponds to a unique vertex in the event order graph.

B. There exists an edge from vertex i to vertex j if

i. Vertex i and vertex j correspond to references generated by the same processor, p, and vertex j is after i, as specified by the program running on processor p.

or

ii. If j is a read which fetched the value written by reference i, or i is a read which fetches the value of the shared location before it is modified by j.

Consider uni-processor systems for a moment. Provided that memory operations are issued in program order, there cannot exist a cycle in the event order graph (if the memory does not permute the order of references). Hence, events are Sequential Consistent. But for multiprocessor systems with private caches, writes are not atomic; therefore issuing memory operations in program order is not a sufficient condition for sequential consistency. Consider the following example (Figure 1.3)

Processor 1	Processor 2
R1(a)	R2(b)
W1(b)	W2(a)

Figure 1.3 Parallel Program

A cycle in the event ordering graph will occur if the code executes in the following way. Processor 1 has a cache miss when it reads a, and the read request is slowed by network traffic. While the read to a is in the network, the write to b is issued by Processor 1 and the invalidations of the other processor's caches happens quickly. After Processor 2 invalidates its cache, Processor 2 reads the new value of b and writes to a. This all happens before Processor 1's read request reaches the memory module. In such an execution sequence, the program will have the an event order graph like Figure 1.4

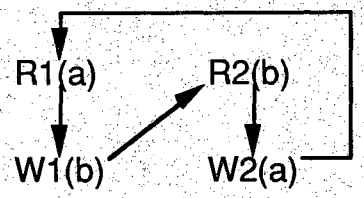


Figure 1.4 Non-Sequentially Consistent Execution of a Parallel Program

This is a very strange kind of behavior for programmers to take into account when writing programs, so two more straightforward event ordering definitions were developed by Dubois et. al. The first definition is called Strong Ordering; it maintains the sufficient conditions for Sequential Consistency in multiprocessors [DuSB86] [DuSB88]. The conditions for Strong Ordering are:

1. Accesses to shared data by any one processor are issued, and performed in program order.

Strong Ordering has a second condition which is needed whenever writes to shared data are not atomic.

2. At the time when a write to shared data by processor i is observed by processor j, all accesses to global data issued by i before the issuing of the write must be performed with respect to j.

Strong Ordering is a very tight constraint. Most all uni-processor systems don't uphold Strong Ordering; rather, they allow for reads and writes to different locations to be permuted, and reads to the same location to be permuted. This is done to improve performance.

A more realistically ordering of events is Weak Ordering [DuSB86] [DuSB88]. Weak Ordering divides references into two large categories: references to synchronization variables and references to all other variables. A synchronization variable is any variable used to indicates to other processors that data is available for reading. They also help to guarantee that there is only one writing processor to a location at a time. For the most part these are semaphore variables. A system is Weakly Ordered if:

1) Accesses to synchronization variables are strongly ordered.

2) No access to a synchronization variable is issued by a processor, p, before all previous shared data accesses issued by processor p have been performed with respect to all other processors.

3) No access to shared data is issued by a processor, p, before an access to a synchronization variable has been performed, with respect to all other processors.

Requirement one is easily maintained by not caching the synchronization variables.

At first, it might not seem that requirement two is essential to the correct operation of parallel programs. Consider a processor which immediately after entering a critical section, reads a shared variable -- `array_index`, used to index into an array of data elements -- and just before exiting the critical section increments `array_index`. If when the processor leaves the critical section it issues a write to array_index, but does not wait until the write is performed with respect to all other processors, another processor with an old copy of `array_index` cached can enter the critical section. Once in the critical section, this new processor might reference `array_index` and receive the old value.

Requirement three requires programs be written so that there cannot be processors trying to read or write to a location which are being written to.

For the remainder of the thesis only Weakly Ordered systems will be considered.

1.7 Multistage Interconnection Networks

Multistage interconnection networks are a compromise between mutually exclusive, inexpensive busses and shareable, expensive crossbars [Sieg85]. Consequently multistage interconnection networks are desirable to build large scale multiprocessors around. The most popular multistage network is the multistage cube, used in the BBN Butterfly [BBN85] and Ultra Computer [GGKM83]. In one pass of the network, a message can go from any port to any other port, and by including an extra stage in the interconnection network, the network can be made to tolerate one faulty box or two faulty links. The multistage cube network has $\log_2 N$ stages of N/2, two by two switch boxes; each box can be set to straight, exchange, upper broadcast, or lower broadcast, determined by the routing tag carried along with a packet. Figure 1.5 shows an eight-ported multistage cube network.

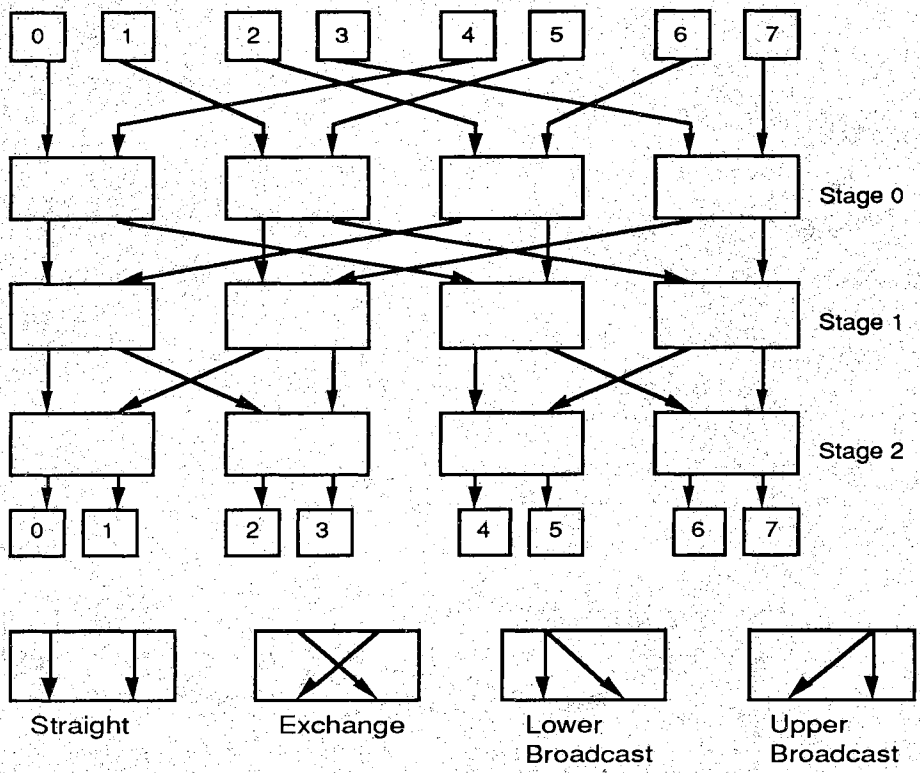
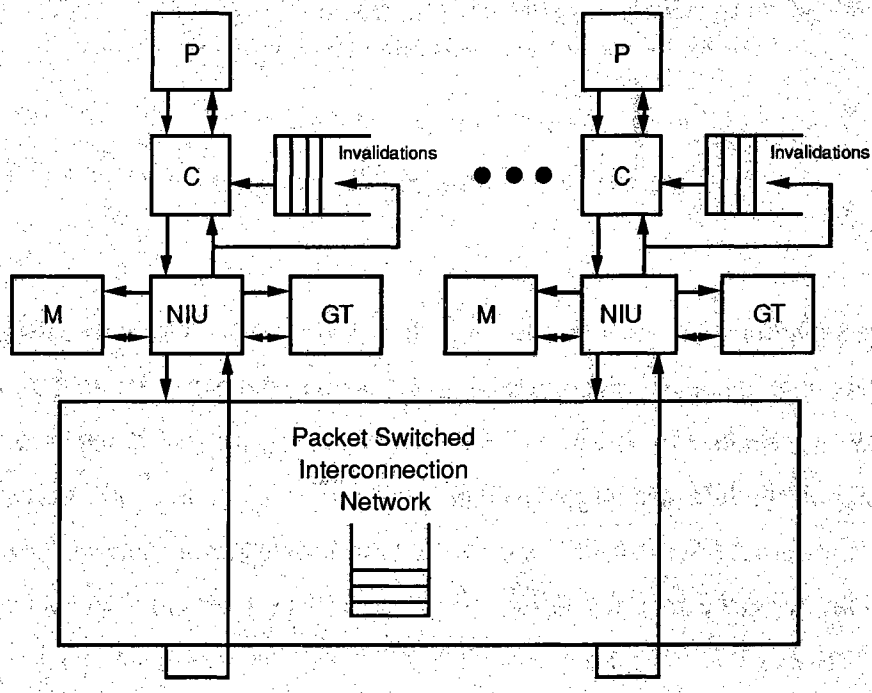


Figure 1.5 Multistage Cube Network

Since the network handles every request every processor generates, there is pressure for this to be as fast as possible. High performance systems will want to use networks with unidirectional links, because arbitration for the links between switch boxes will slow the network down considerably. If a processor shares its network port with a memory module, only one, unidirectional multistage cube network is needed. Figure 1.6 shows this organization which will be referred to as the "Home Memory" configuration. It gets its name because references, made by a processor, don't need to use the network if the memory location resides in the module attached to the same port as the processor.



NIU-Network Interface Unit

Figure 1.6 Home Memory Configuration

Included with each message in a multistage cube network is a routing tag specifying the settings for each of the switch boxes the message goes through. There are two classes of routing tags: single destination and multiple destination. The destination of a message serves as a simple and efficient single destination routing tag, requiring only $\log_2 N$ bits. Multiple destination tags can be used to send the same message to several processors. This is useful for invalidating the other processors which have the block cached when write hits or write misses occur. Several multiple destination schemes have been presented. One method is to use an N bit vector as the routing tag. Bit i of the vector is set if port i is intended to be a destination. In order to represent any arbitrary set of destination ports, all N bits are required. A more concise, multiple destination routing tag is the broadcast mask [Seig85] consisting of a $\log_2 N$ bit routing vector and a broadcast vector of equal length. The routing vector is any one of the destinations of the message, and the broadcast vector indicates which stages should perform broadcasts. A box in stage i looks at bit i for the broadcast vector, and if the bit is asserted the switch box is set to broadcast; otherwise, it looks at the routing

vector to determine whether the switch box should be set to straight or exchange. For example, if N=4 and one wants to send a message to port zero and two, the broadcast mask could be {routing vector=00_2; broadcast vector=10_2} or X0 for short. Notice when the destinations are zero (00_2) and three (11_2), the present set cannot be represented with one broadcast mask.

Because each switchbox acts independently of other switchboxes, no guarantee can be made about the arrival order of two message sent from different ports to the same destination port, but we will assume that two message sent from the same port to the same destination port will arrive in the order they were issued.

1.6 Multiple Channel Architecture

In the more remote future, optical busses may be feasible for processor to memory interconnection. Wailes and Meyer are beginning work on a frequency multiplexed optical bus they call Multiple Channel Architecture [WaMe90]. The bus will have as many channels as discernable frequencies of light (likely to be several thousand). The performance improvement comes because each channel can be used concurrently with every other, and hence greatly reducing interconnection network contention. Although the interconnection is physically a bus, the snoopy protocols will perform poorly in the multiple channel architecture, because the snoop unit at the private caches would have to "snoop" every channel at the same time. On the other hand, directory based protocols can utilize the many independent channels.

2. ANALYSIS OF SHARING

In order to evaluate the effectiveness of coherency protocols it is useful to know some properties of shared cache blocks. Some of the questions which we want to ask are: What percentage of blocks are shared blocks in a typical program? How many processors are likely to share a cache block? How many references does a processor typically make to a block before another processor writes to it?

2.1 Amount of Sharing and Frequency of Writes

From the analysis done on parallel applications [WeGu89] [BaRa89] [EgKa88] [ASHH88], typical values for the fraction of references to shared variables and the fraction of writes can be known. The fraction of references to shared variables ranged from 1.98% to 21.5%, and was on average 10% for the application observed. The fraction of writes ranged from 7% to 40%, and was typically 30%.

2.2 Markov Chain Models of Shared Blocks

To answer the question: "How many processors are likely to have a shared block cached?", a Markov chain was developed -- one that transitions from state to state whenever a private cache issues a global table operation.

A Markov chain is a fitting model for the behavior of the shared block because the future state of the block only depends upon the present state. This is the fundamental Markov property. Other assumptions which need to be made are:

1) Every processor is equally likely to cache any one of the shared blocks.

2) The accesses to shared blocks are uniformly distributed across all the shared blocks.

3) Every block in a private cache is equally likely to be ejected.

4) Each memory reference is independent of all other references.

2.2.1 Markov Chain for RW Blocks

The Markov chain for a representative shared RW block is presented in Figure 2.1. The states a shared RW block can be in are Not Cached, Cache with RW privilege by one processor, or cached with RO privilege by any number of processors. This chain is similar to the Markov chain present by Dubois [Dubi87], but this chain incorporates block ejections.

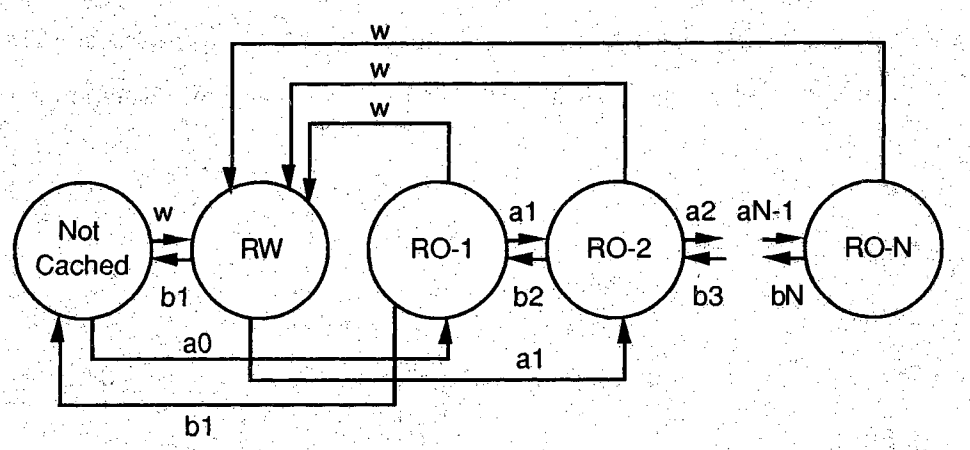


Figure 2.1 RW Data Markov Chain

The transitional probabilities can be expressed in term of "observable" properties of the system and program, as done for bus snooping protocols by Yang et. al.[YaBL89].

$$w = \text{write probability} \quad [1\text{-qro}][qs][Fw]\left[\frac{1}{Nsb}\right] \tag{2.1}$$

$$ai = \text{probability of adding a processor} \quad [1\text{-qro}][qs][1\text{-Fw}]\left[\frac{N-i}{N}\right]\left[\frac{1}{Nsb}\right] \tag{2.2}$$

$$bi = \text{probability of deleting a processor} \quad [1\text{-h}]\left[\frac{i}{N}\right]\left[\frac{1}{Ncb}\right] \tag{2.3}$$

Definition of parameters:

Ncb = Number of blocks in a single private cache

Nsb = Number of shared cache blocks in an application

qs = Fraction of references which are to shared data

qro = Probability that a reference is to a shared RO block given that the reference is to a shared block

Fw = Fraction of references which are writes

h = Hit ratio of the private caches

The steady state probabilities for any Markov Chain can be solved by finding the solution to the set of N+3 equations below:

$$\Pi = \Pi T \qquad \text{(N+2 equations)}$$

$$1 = \left[\sum_{i=0}^{N+1} \pi_i \right] \qquad \text{(1 equation)}$$

Where:

T = (N+2)x(N+2) transitional probability matrix

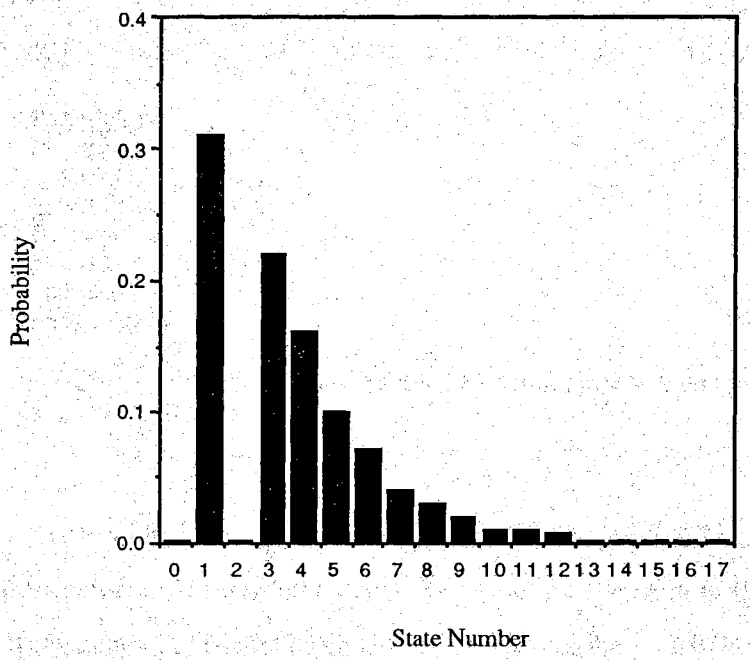
Π = (N+2)x1 steady state probability matrix

π_i = steady state probability of state i

Once the π_i's are known, the expected size of the present set (|P|) can be calculated.

$$E[|P|] = \pi_1 + \sum_{i=2}^{N+1} (i-1)\pi_i \qquad (2.4)$$

Using numerical analysis software, the steady state probabilities for a sample system were solved for.

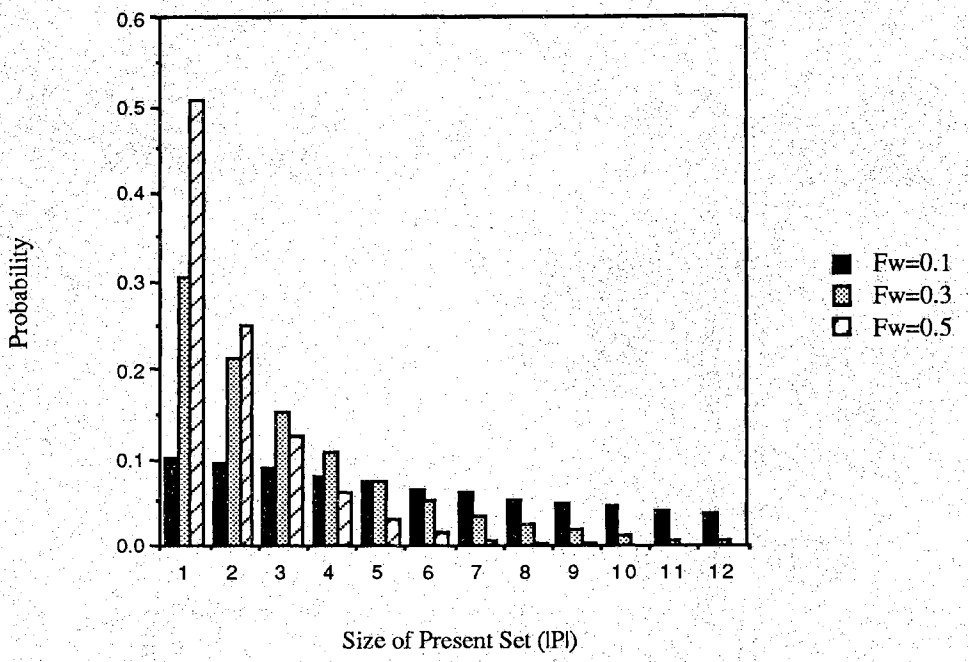


Parameters
N = 32
qro =1.0
qs = 0.2
Fw = 0.3
h = 0.9
Nsb = 64
Ncb = 4096

Figure 2.2 Steady State Probability of RW Markov Chain

Because the assumed hit ratio is ninety percent and Nsb/Ncb is a small fraction, ejections are infrequent, and consequently shared RW blocks are seldom in the Not Cached state or RO-1 state. If the number of shared blocks increases, so does the probability of these states.

The steady state probabilities are most affected by Fw, the fraction of writes. As Fw increases, the expected size of the present set decreases.



Parameters
N=32
qro = 0
qs = 0.2
h = 0.9
Nsb = 64
Ncb = 4096

Figure 2.3 Effects of Fw on Steady State Probabilities

This Markov chain's steady state probabilities reflect the number of invalidations required to perform a write. The results we obtained concur with the number of invalidations per write observed by Weber and Gupta [WeGu89] as well as Agarwal et. al. [ASHH88] in their analysis of parallel applications

The validity of assumption number one: every processor is equally likely to cache any one of the shared blocks might be questioned, because frequently in parallel applications not every processor references every shared location. But, the number of processors which reference a block does not alter the Markov chain more than to change N to a number which better suits the

application of interest. A more realistic model would be to divide the shared blocks into several different classes according to how many processors access the various shared block.

2.2.2 Markov Chain for RO Blocks

Read only (RO) data which remains RO for the entire execution of the program, like instruction data, should be marked as local by a compiler, and lumped into local pages by the operating system. Once a processor performs address translation, the global or local distinction is known, and local cache miss operations will require coherency maintenance. Commercial systems like the 88000[Moto88] use this strategy to reduce the amount of traffic to the snooping units. This strategy also helps reduce the traffic to the global table.

Even with intelligent compilers and operating systems which put local data into distinct pages, some shared data is global and RO. In VLSI channel routing for example, the vertical and horizontal constraint graphs are RW during the phase which generates the graphs [WaCa89], then during the phase where the nets are assigned to a track, they become RO.

A Markov model can be constructed for RO data. It is a degenerate case of the RW Markov chain where the write percentage is zero (Fw = 0).

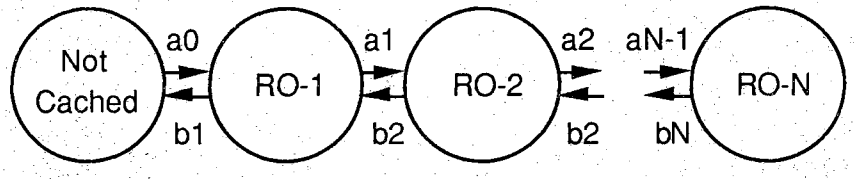


Figure 2.4 RO Data Markov Chain

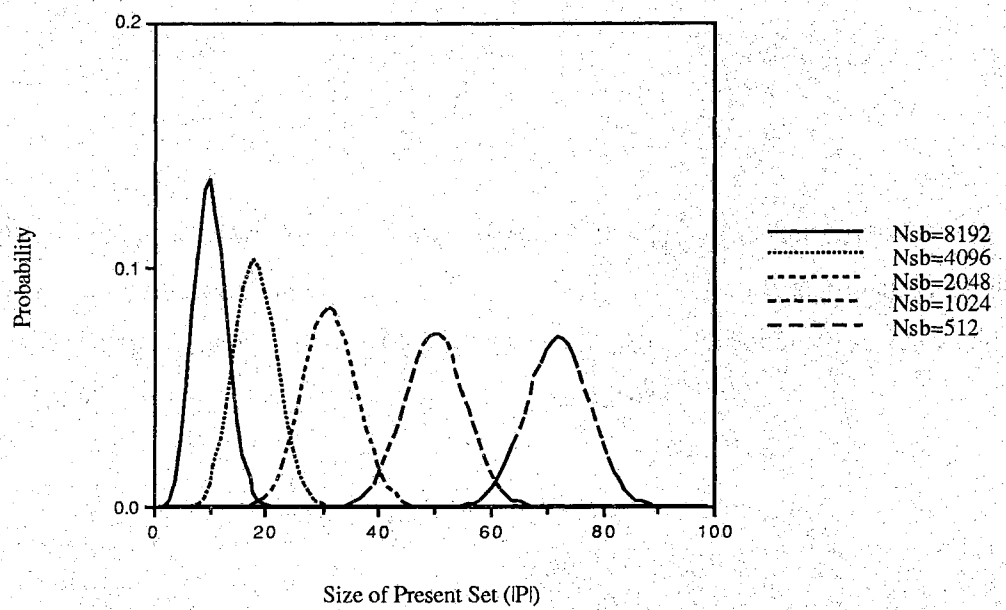
$$a_i = \text{probability of adding a processor} \left[\frac{(N-i)}{N}\right][qs][qro]\left[\frac{1}{Nsb}\right] \quad (2.5)$$

$$b_i = \text{Probability of removing a processor} \left[\frac{i}{N}\right][1-h]\left[\frac{1}{Ncb}\right] \quad (2.6)$$

The closed form solution of the steady state probabilities for the RO Markov chain is:

$$\pi_i = \left[\frac{\left[\prod_{j=1}^{i} \frac{a_{j-1}}{b_j} \right]}{1 + \sum_{k=2}^{N} \prod_{j=1}^{k} \frac{a_{j-1}}{b_j}} \right] \quad (2.7)$$

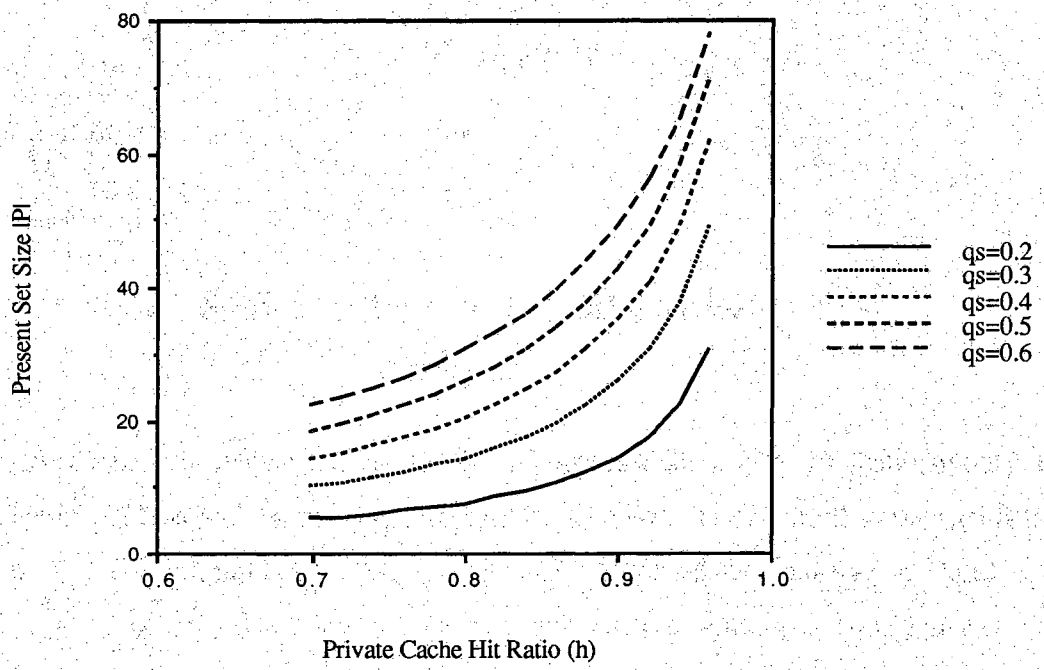
This is unlike the RW case where the effects of Fw seemed to dominate, here all the parameters play a significant role in influencing the steady state probabilities. Equation 2.6 does not offer intuitive understanding as to how each parameter effects the probability density function of |P|, so several graphs were generated for various system parameters.



Parameters
N = 128
qro = 0.5
qs = 0.5
h = 0.9
Ncb = 32768

Figure 2.5 Steady State Probability for RO Markov Chain

As the number of shared blocks increases, the likelihood of a block being cached by more than one processor diminishes. The cardinality of the present set also is affected by the hit ratio of the caches and the percentage of shared references.



Parameters
N = 128
qro = 0.5
Nsb = 2048
Ncb = 32768

Figure 2.6 Effects of Hit Ratio and qs on Present Set

As the hit ratio and the percent of shared references increases, the cardinality of the present set increases. This is because the only effect which decreases the size of the present set is block ejection. When the hit ratio is low, block ejections are less frequent. When qs increases, shared blocks are accessed more frequently, making the likelihood of a block being cached greater.

2.3 Frequency of References to Shared Blocks

It is useful to know how many references a processor will make to a block that it wrote to before another processor writes to it. This gives an indication of the utility of caching dirty blocks. This, of course, is program dependent. Baylor and Rathi [BaRa89] analyzed the behavior of several engineering and scientific applications, and reported a measure corresponding to the amount of time after a processor writes to a block and before another processor writes to that same block. The unit of time they used was a logical cycle. In a logical cycle, every processor can perform a read or a write to memory. When the block size was four words (the best and smallest size in their analysis), the average number of cycles was in the hundreds. Eggers and Katz, in their analysis of snoopy bus systems, monitored the number of writes a processor made to a block after it's first write miss or write hit and before another processor wrote to the block. They called this the write run length. The number of reads and writes to a shared block during this time can be estimated if the write run length and the fraction of writes are known.

$$\text{Read and Write Run Length} = \text{Write Run Length}/Fw \qquad (2.8)$$

Using the numbers gathered by Eggers and Katz, the read write run length for the four applications observed ranged from 6 to 22, and on the average was 13.

3. FIXED LENGTH ENTRY DIRECTORIES

3.1 Traditional Directory Based Protocols

Chronologically, Tang's method [Tang76] is the first directory based cache coherency protocol. He proposed to store a copy of the private cache tags in M, K*N-way associative memories each Ns/M sets large (where N is the number of processors in the system, M is the number of memory modules, Ns is the number of sets in the private caches, and K is the associativity of the caches). The major drawback to such an approach is the access time of the associative memory. In most logic families, the access time for a CAM (Contents Addressable Memory) is directly proportional to the associativity. This property of associative memory makes Tang's approach hard to scale. We describe a way to design out the associate memory in the next chapter, but when done, the memory size is comparable to the amount of memory required by the next approach we will now discuss.

Censier and Feautrier [CeFe78] developed a coherency protocol that does not require associative memory. Instead, the global table contains an entry for every block in main memory, each entry being N+1 bits. Bit zero of the entry records the type of access privilege the cache(s) with the block have (Only one cache may have read and write access to a block at a time, but all caches may simultaneously cache the block if all have read only access) If this privilege bit is asserted, a cache has read and write privilege (RW); otherwise, the cache(s) have read only privilege (RO). The next N bits determine which of the private cache(s) have the block cached. If processor i has block b cached, then bit i of the global table entry for block b is asserted. This N bit array is referred to as the **present vector.** If a block is not cached, the present vector is all zeros.

Whenever a read miss, write hit, write miss or block ejection occurs at one of the private caches, a request is sent to one of the memory modules based on the address of the reference. How the global table is updated depends upon what type of operation being performed, and what state the block is currently in. The actions for each of the five basic operations are:

Read Miss (By processor i to block b)

RO

Bit i of the entry is set, and the block is fetched from main memory.

RW

An **invalidate without intent to modify** message is sent to the processor which has RW privilege to the block; call it processor j. Processor j sends the block to main memory and changes his local state to RO. The final global table entry is RO with bits i and j set. While the block is being written back, the entry for the block must be put in a "pending" state. If another processor has a read miss on the location while the entry is in the "pending" state, that processor should be recorded as requesting the block. It is assumed that program synchronization will make it so no write misses occur while the entry is in the pending state. When the block arrives at main memory, the global table is updated, and a copy of the block is sent to the requesting processor(s).

Not Cached

The entry is set to RO with bit i set, and the block is fetched from main memory.

Write Hit (By processor i to block b)

RO

An **invalidate with intent to modify** message is sent to processor i if and only if i is an element of P. The final entry is set to RW, with only bit i in the present vector set. The block sent to the requesting processor is a copy of what resides in main memory.

RW

The write can be performed locally without any interconnection network messages.

Write Miss (By processor i to block b)

RO

An **invalidate with intent to modify** message is sent to processor i if and only if i is an element of P. The final entry is set to RW, with only bit i in the present vector set. The block sent to the requesting processor is a copy of what resides in main memory.

RW

An invalidate with intent to modify message is sent to the processor which has RW privilege to the block. The final entry is set to RW privilege with only bit i set. The processor with the dirty block sends it to the requesting processor. It is assumed that the program synchronization will assure that no other write miss to this block will occur until after the dirty block arrives at the processor which had the write miss.

Not Cached (Write Miss only)

The entry is set to RW with bit i set, and the block is fetched from main memory.

Ejection Request (Of block b by processor i)

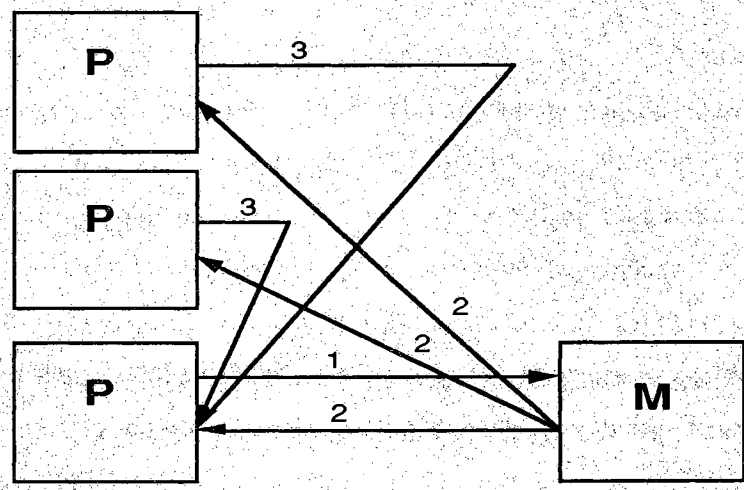
RO bit i is cleared in the entry.

RW bit i is cleared from the entry, and the block is written to memory.

3.2 Maintaining Weak Coherency

In order to maintain Weak Ordering, a system must insure that all accesses which are issued before the issuing of a synchronization variable access, are "performed" with respect to all other processors. A convenient way to enforce this is to have two counter for each processor. The first counter is a called a **return receipt** counter which records the number of return receipts which have been received. This counter is incremented every time a read to a shared block is issued and decremented every time the read data is returned. Writes are more complicated. A write is performed with respect to all processors when all the other processors which have the block invalidate it, and the global table is updated. When a processor issues a write operation, the **outstanding writes** counter (the second counter) is incremented. The counter is decremented when it receives a return message from the memory module indicating the number of invalidation

messages which were sent as a result of the write. This number must be added to the return receipt counter. Thus, this counter records the number of writes which have been issued, but have not updated the return receipt counter. Since it is entirely possible for some return receipts from the invalidating processors to be received before the message from the memory is received, the return receipt counter must be designed to hold both positive and negative numbers. When both the outstanding writes and the return receipt counter are zero, all references are performed with respect to all other processors, and accessing synchronization variables is allowed. Consider the example write miss illustrated in Figure 3.1.



1) Outstanding Writes counter incremented when the write is issued
2) Global table returns the size of the present set
   - Return Receipt counter += size of the present set
   - Outstanding Writes counter decremented
   Other processors which have the block are sent invalidation messages.
3) The invalidating processors send return receipts
   - Return Receipt counter decremented for each message

Figure 3.1 Two Counter Example

As Brooks and Hoag [BrHo90] mention, a facility like this makes it possible for normal program variables to be used as synchronization variables. By simply surrounding the access of the variable by calls to an operating function wait(), the variable can be treated as a synchronization variable. The wait() function stops the processor from issuing any more references until both counters are zero.

3.3 Evaluation of Traditional Protocols

For a system which uses the Censier and Feautrier protocol to maintain cache coherency, the global table can become very large. A system with 64 processor, 256 Megabytes of main memory, and a block size of 16 bytes, will need a 130 Megabyte global table -- 50% the size of main memory! The size of the global table is not large because the entries are inefficiently representing the possible states of a shared block. Since any combination of processors can simultaneously cache the block with RO privilege, there are 2^N, RO states, and since each processor can obtain RW privilege to the block, there are N, RW states. N+1 bits are capable of recording 2^{N+1} states, so the efficiency of this global table entry format is:

$$\text{Efficiency} = (2^N + N)/\,2^{N+1} \qquad (3.1)$$

$$\text{Lim Efficiency} = 0.5$$
$$N \to \infty$$

This implies that an alternate fixed entry format capable of recording every possible state will not yield an order of magnitude improvement.

The most obvious way of reducing the global table size is to increase the block size. For every doubling of the block size, the global table size is halved, because only one entry is needed for each block in main memory. But increasing the block size degrades system performance when the block size is made very large. Often, unnecessary words are carried along with other references when the block size is big [DuBr82].

An economical protocol [ArBa85] has been proposed which only requires two bits per block, consequently each block can be in one of four states: Modified, Present, Present*, or Not Cached. The entry is set to Modified if a cache has a dirty copy of the block. If one of the caches has the block RO, then the entry is set to Present. If more than one processor has the block cached, the entry is set to Present*. The distinction between Present and Present* is made so no invalidation messages need to be sent when write hits occur to blocks which are in the Present state. Whenever one or more private caches need to be invalidated, an invalidation message has to be sent to every processor. Because the broadcasts are to all the processors in the system, excessive, unnecessary network traffic is introduced, and each private cache must spend time servicing invalidation requests for blocks which are not present.

3.4 Criticisms of Traditional Techniques

The major criticisms of the traditional global table cache coherency protocols are:

1) The amount of interconnection network traffic is great.

2) The amount of memory required to make the global table is great.

3) The number of unnecessary invalidation to the private caches distracts them from serving the processor they are privy with.

The remaining sections of the thesis will be devoted to grappling with these problems, by inventing new protocols and altering global table architectures.

3.5 Using Broadcast Masks as Global Table Entries

A compromise between the economic protocol and the Censier and Feautrier approach is to record the present set with a single broadcast mask. Equation 3.2 expresses the size of the global table when this entry format is employed.

$$\text{Global Table Size (with broadcast masks)} = M*(2*\log_2(N)+1) \text{ (bits)}$$
$$= O(M\log_2 N) \quad (3.2)$$

M is the number of blocks in main memory.

As shown in Figure 2.2, the most likely combinations for shared read/write data are those combinations where $|P| < (1/5)N$, so the global table entry format only needs to be accurate when several processors have a block cached.

Because not every combination of processors can be represented with a single broadcast mask, processors which are not in the present set may be inadvertently included in the broadcast mask. In fact, the processor which performs an operation which invalidates all other copies of the block may itself receive an invalidation message which it must ignore. In order for the processor to discern whether or not an invalidation message should be disregarded, the invalidate operation is divided into two types: invalidate (RW) and invalidate (RO). The global table sends out invalidations consistent with the state it has recorded, and if a processor receives an invalidation inconsistent with its record of the access privilege, the invalidation is disregarded.

Assuming that every processor is equally likely to cache a block, a simulator was built to determine how accurately a broadcast mask can represent the present set.

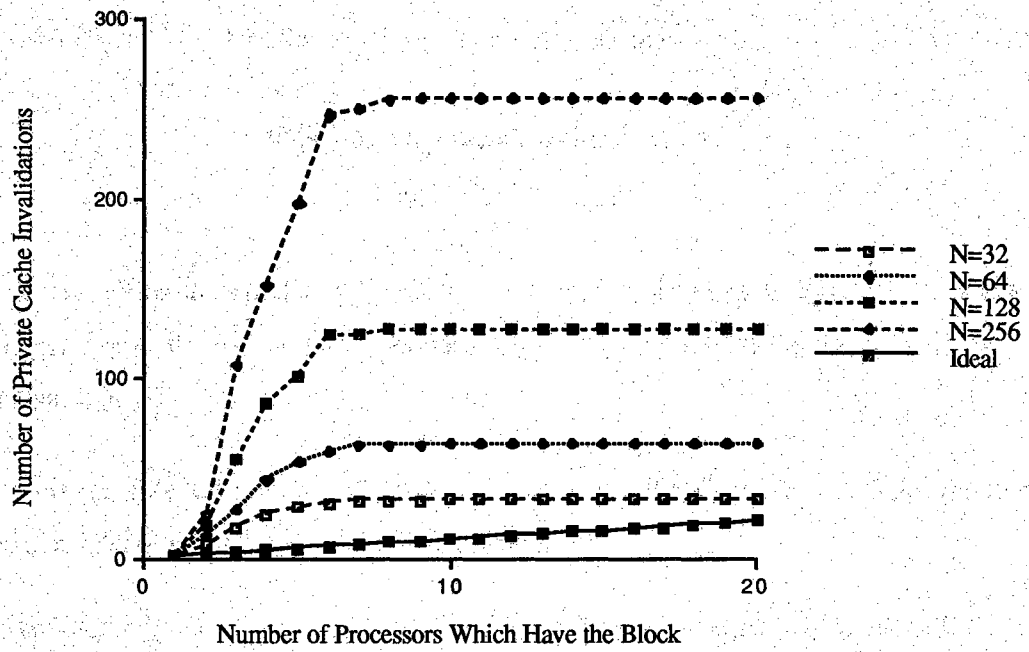


Figure 3.2 One Broadcast Mask System

Evident from the simulation results, the single broadcast mask method degenerates to the performance equal to that of the Archibald and Baer economical solution when $|P| > 10$, independent of N.

Representing the present set with several broadcast masks reduces the number of extraneous invalidations, but there are some complications.

We present three theorems showing the difficulties in preventing extraneous and redundant invalidations. Theorem 3.1 is an upper bound on the number of broadcast masks needed; 3.2 is

the corresponding lower bound. Theorem 3.3 shows the difficulty of determining which masks should cover which processors.

3.5.1 Theoretical Difficulties

Theorem 3.1: (An upper bound on the number of broadcast masks)

The maximum number of broadcast masks needed to represent an arbitrary P without introducing extraneous or redundant invalidations is N/2.

Example:

Consider a system with N=8, and a particular block where P = $\{001_2,010_2,100_2,111_2\}$. Because every processor number is a Hamming distance of two away from every other processor number, none of the processor numbers can be merged into a broadcast mask without introducing extraneous invalidations. Notice |P| = N/2 = 4.

Lemma 3.1:

Let S be a set numbers such that, for every number in the set, there does not exist another number in the set which is a Hamming distance of one away. The minimum number of bits to represent any S is $\log_2(|S|) + 1$. This is a the idea behind single bit parity.

Lemma 3.2:

The maximum number of broadcast masks ever needed equals the cardinality of S (|S|).

Proof by contradiction:

Assume there exists a set S2, such that |S2| > |S|. Let O=S2-S. Every element in O is a Hamming distance of one away from an element in S (otherwise, the element would be in S). Hence, every element in O can be covered by altering one or more of the broadcast masks, without introducing extraneous or redundant invalidations. Thus no more than |S| broadcast masks are required.

Proof of theorem:

Because each processor's ID is $\log_2 N$ bit long,

$$\log_2(|S|) + 1 = \log_2(N)$$

$$|S| = 2^{(\log_2(N)-1)} = N/2$$

maximum number of broadcast masks needed = |S| = N/2

Even if the present set is the best case (the combination of processors which requires the fewest number of broadcast masks), several broadcast masks may be needed to represent it.

Theorem 3.2: (A lower bound on the number of broadcast masks)

The lower bound on the number of broadcast masks needed to represent the present set (without introducing in extraneous or redundant invalidations) is Hamming(|P|,0).

Proof:

Broadcast masks are only capable of having destination sets of size 2^j; $0<=j<=\log_2 N$. Each asserted bit in |P| represents a group of size 2^i, where i is the bit's position. Since no collection of groups can be combined into one group without changing |P|, the minimum number of broadcast masks must equal Hamming(|P|,0).

The last major difficulty with representing the present set with multiple broadcast masks is the complexity of trying to determine which masks should cover which processors.

Theorem 3.3: (Optimal Covering in NP Complete)

Given a present set, determining the minimal set of broadcast masks which does not introduce any extraneous or redundant invalidations is an NP complete problem.

Proof:

The problem is polynomially related to the logic minimization problem [BHMS85]. Showing that another problem is polynomially related to a problem which is known to be NP Complete proves that the new problem is also NP complete [Baas78]. If there exists a polynomial time algorithm to convert the a solution of the optimal covering problem into a solution for the logic minimization problem, optimal covering is NP complete. Both the logic minimization problem and the optimal covering problem have the same input: a set of minterms which need to be covered. Both problem's solutions are stated in term of prime implicants, but not necessarily the same prime implicants. In the optimal covering problem the prime implicant must be disjoint, and in the logic minimization problem they are to overlap as much a possible. A polynomial time algorithm to convert an optimal covering solution to a logic minimization solution is given in Figure 3.3. The algorithm assumes that the solution to the minimum number of masks problem is stored in an array of mask called PI (prime implicants). Each prime implicant (PI[j]) has $\log_2 N$ trinary digits (0,1, or X); digit b of mask j is indicated by PI[j]:b.

```
for i=1 to number_of_PI's do begin
   for b=0 to log2N-1 do begin
      if(PI[i]:b <> X) then
         if PI[i] with PI[i]:b = X only covers wanted minterms
            PI[i]:b = X;
   end
end
```

Figure 3.3 Mapping of Minimal Broadcast Problem to Logic Minimization

Consider an example where the where N=16 and the minterms which need to be covered are: 4,5,6,7,13,14, and 15. Figure 3.4 shows these minterms placed on a Karnaugh map, the solution to the optimal covering problem, and the corresponding logic minimization problem.

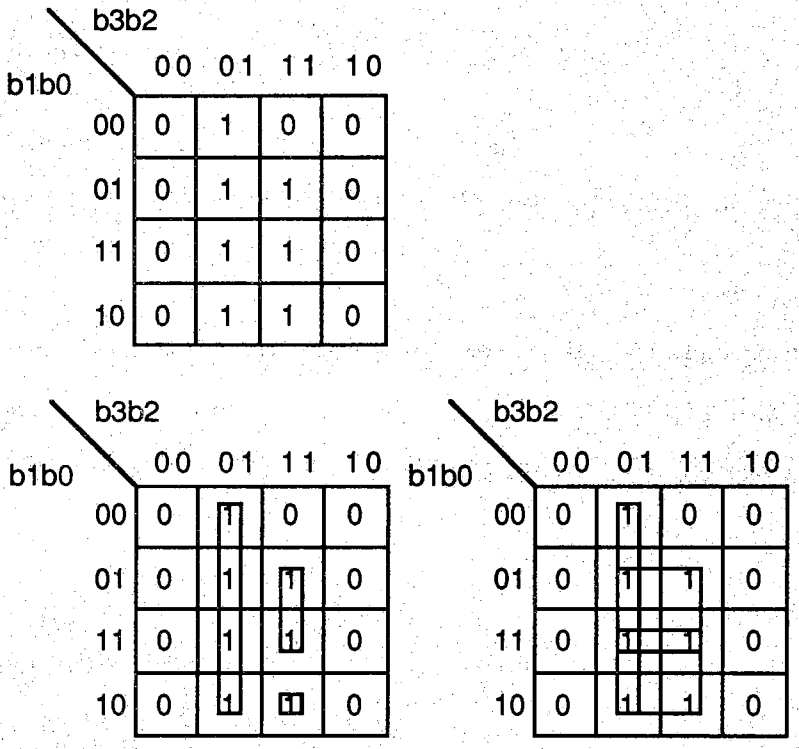


Figure 3.4 An Example Covering

The number of minterms which need to be covered in the size of the inputs (n). In the worst case, the number of PI's equals n, and for each one of the iterations of the outer loop, i, the algorithm has to compare with every other minterm. Consequently, the above algorithm is $O(n^2)$.

How do we know that the solution to the logic minimization problem generated above, is indeed a proper solution? We know that the number of prime implicants is the minimum number, since the input to the conversion algorithm solution to the optimal covering problem.

The optimal covering problem is not only encountered when new processors need to be added into broadcast masks in broadcast mask systems. The same problem arises if the entry format is Censier and Feautrier's and interconnection network uses broadcast masks for routing messages.

Because of the difficulties of eliminating extraneous and redundant invalidations, a multiple broadcast mask scheme which introduces redundant and extraneous invalidations was developed. Initially, all the broadcast masks for each global table entry are invalidated. When a processor which is not currently covered by any mask caches the block, the processor number is compared with each broadcast mask, and merged it into the broadcast mask to which it is closest. Distance between a processor number and a broadcast mask is defined as:

Distance = Hamming(0, (broadcast vector & (processor number ^ routing vector))

When two masks are the same distance from a processor number, the one with the fewest number of asserted bits in the broadcast vector is chosen, and if this fails to resolve a conflict, one of the several, closest masks is arbitrarily chosen.

This method does not guarantee to minimize the number of extraneous or redundant processors. Consider the two broadcast masks 1XXXX and 1100X in a two mask system, with the new processor being processor 00000. Notice 00000 is closer to 1XXXX than 1100X, though merging with 1XXXX will introduce 15 extraneous invalidations, while merging 00000 with 1100X will only introduce 5 extraneous invalidations.

3.5.2 Simulation Results for Multiple Broadcast Masks

The effectiveness of this method was studied using a simulator which assumed every processor is equally likely to reference a block. Figure 3.5 indicates the precision obtained for various system sizes.

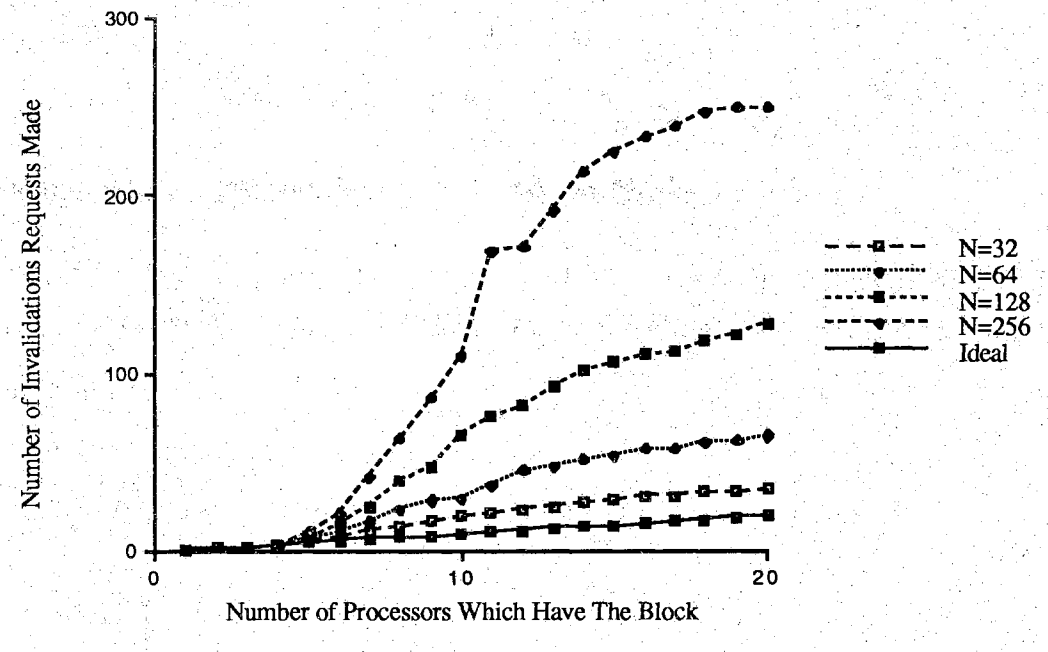


Figure 3.5 Four Broadcast Mask System

The effects of varying the number of broadcast masks (BMS) can be seen in Figure 3.6, for a system with 128 processors.

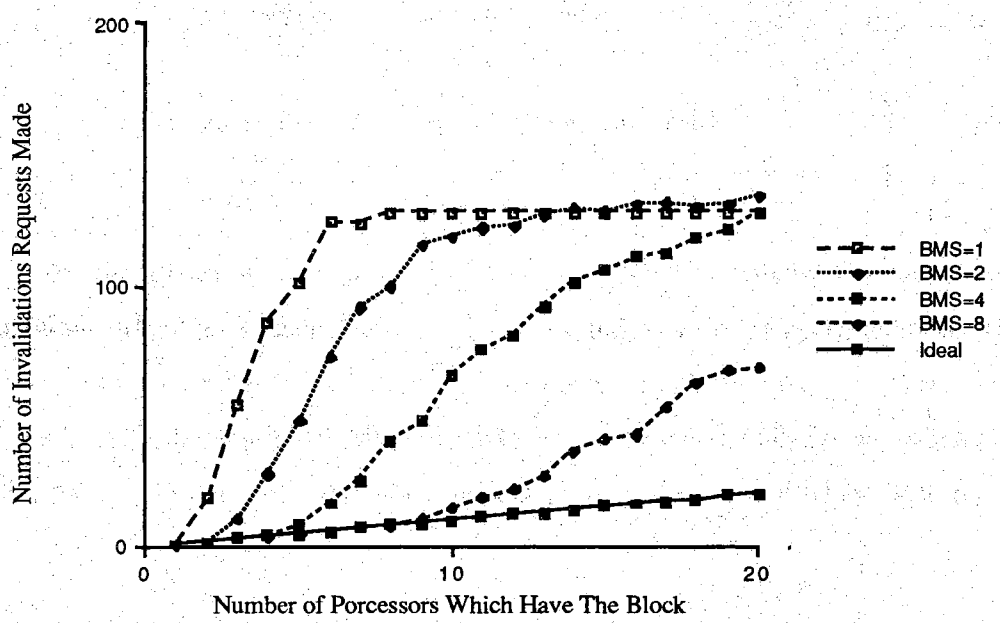


Figure 3.6 Effects of Using More Broadcast Masks

A drawback to the multiple broadcast method is its frequent inability to reduce the number of invalidations when block ejections occur. Seldom can a broadcast mask be reduced. The two cases where a broadcast mask can be reduced are when the ejecting processor is covered by a broadcast mask containing just that processor, or by a mask containing that processor and only one other processor. Consequently, the global table should not be notified when clean blocks are ejected.

3.6 Sloppy Ejection

Before we can bring the Markov models for shared blocks into the analysis of multiple broadcast mask systems, we need to introduce the concept of **sloppy ejection**. Sloppy ejection is not updating the global table when clean blocks are ejected from the private caches (the traditional approach we call **tidy ejection**) This reduces the number of references to the global table, and reduces contention for the table. Traditional directory protocols keep the global table "up to date"

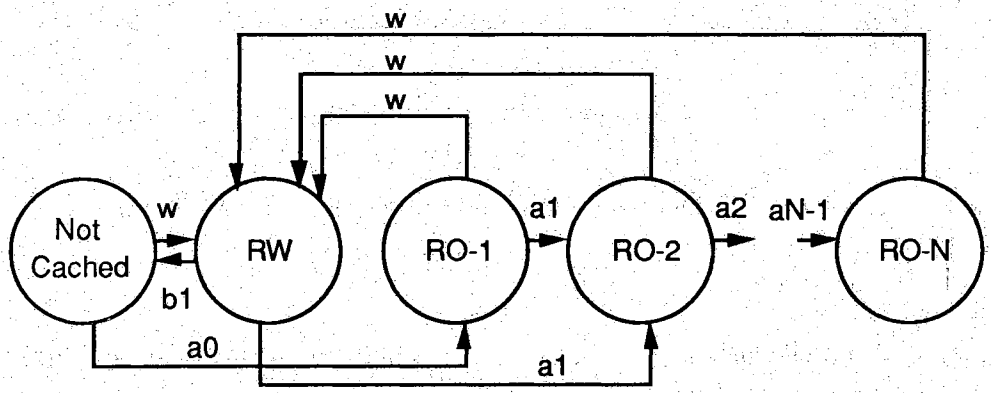


Figure 3.7 Sloppy Ejection Markov Chain

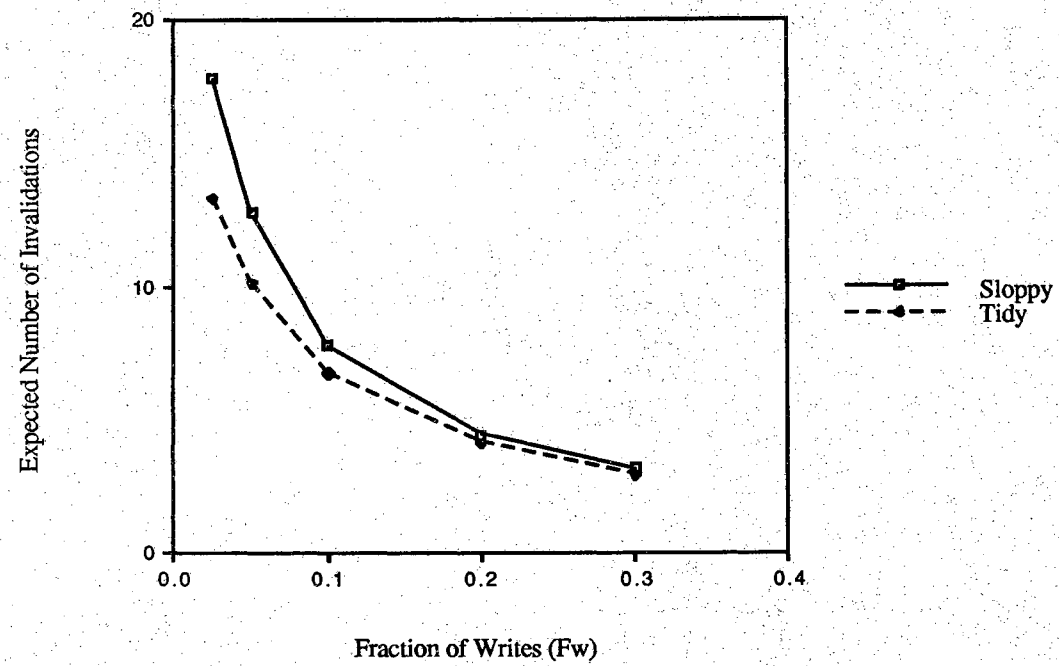
The transitional probabilities are:

$$w = \text{write probability} \quad [1\text{-qro}][qs][Fw]\left[\frac{1}{Nsb}\right] \tag{3.3}$$

$$ai = \text{probability of adding a processor} \quad [1\text{-qro}][qs][1\text{-Fw}]\left[\frac{N\text{-}i}{N}\right]\left[\frac{1}{Nsb}\right] \tag{3.4}$$

$$bi = \text{probability of deleting a processor} \quad [1\text{-h}]\left[\frac{i}{N}\right]\left[\frac{1}{Ncb}\right] \tag{3.5}$$

From the steady state probabilities, E{number of invalidates with tidy ejection} and E{number of invalidations with sloppy ejection} for a systems which uses a N+1 bit vector for its entry format. were calculated to help determine the effectiveness of sloppy ejection. Figure 3.8 compares the two methods for a range of write percentages (Fw) on a system where N=32.



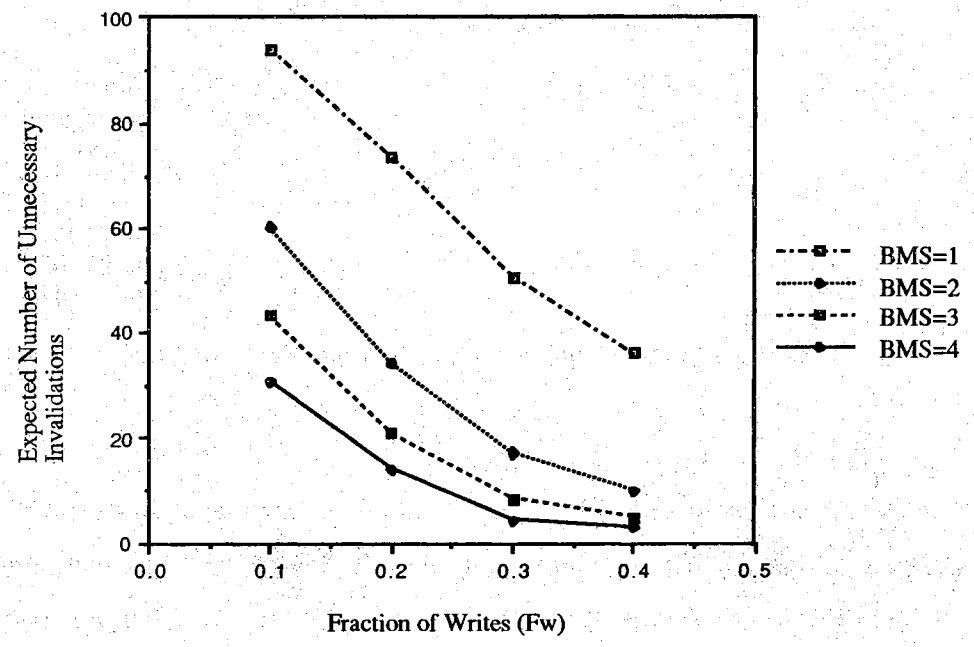
Parameters
N = 32
qro = 0
qs = 0.2
Fw = 0.3
h = 0.9
Nsb = 64
Ncb = 4096

Figure 3.8 Comparison of Sloppy vs. Tidy Ejection

3.7 Expected Number of Invalidations

Using the steady state probabilities from the Markov chain (Figure 3.7) and the broadcast mask simulation results, the expected number of redundant and extraneous invalidations for an invalidate operation can be determined by the law of conditional expectation (3.6).

$$E[\text{unnecessary invaidations}]=\sum_{i=1}^{N} (E[\text{invalidations}| |P|=i]-i)*P[|P|=i] \quad (3.6)$$



Parameters
N = 128
qro = 0.2
qs = 0.2
h = 0.9
Nsb = 512
Ncb = 2048

Figure 3.9 Expected Number of Unnecessary Invalidations

When using more than one mask to represent the present set, the maximum number of invalidation messages can exceed N, because some of the mask will overlap one another generating two or more invalidations to the same processor. Figure 3.10 shows the results from our simulator for various numbers of broadcast masks in a system with 128 processors.

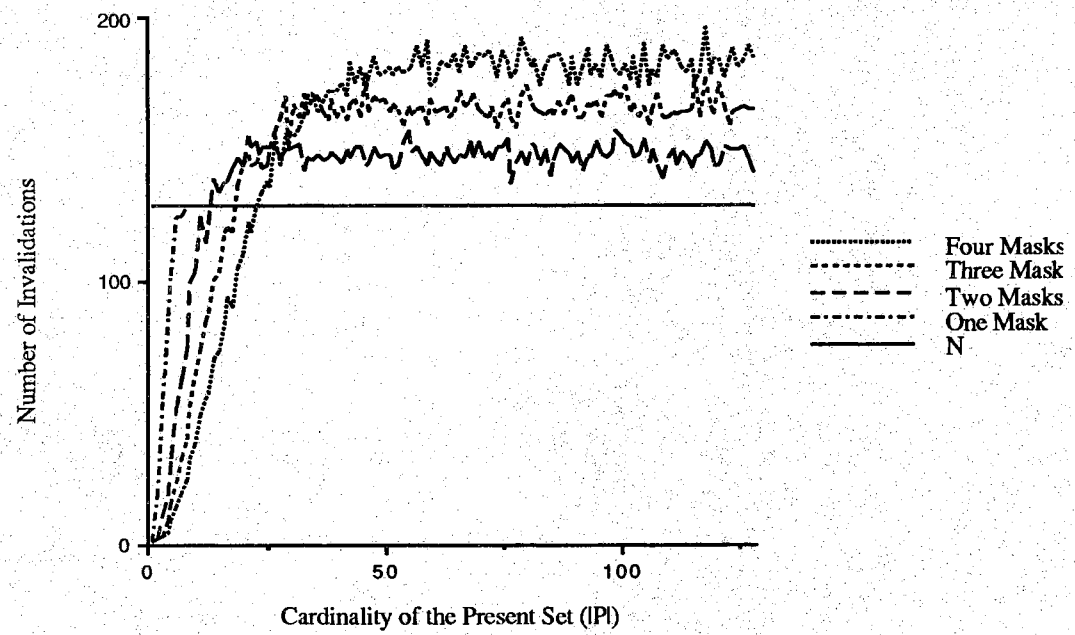


Figure 3.10 Full Range Behavior of Multiple Broadcast Mask Systems (N=128)

The number of processors which are being invalidated (|D|) when the accuracy levels out we call the saturated destination set. The size of the saturated density set roughly follows equation 3.7.

$$|\text{saturated } D| = (0.15)*N*(BMS-1) + N \qquad (3.7)$$

BMS - number of broadcast masks

The simulator was amended, so that whenever a new processor is merged into a broadcast mask, each broadcast mask is compared with every other broadcast mask. If any one of the masks was a subset of another, the smaller mask was invalidated. Masks seldom became subsets of other masks, so this did not make a noticeable difference in the number of invalidations. Not to mention, implementing such a property would be costly.

If many broadcast masks are used to represent the present set, the number of bits required for each block will exceed that required by the present vector approach. The maximum number of broadcast masks which can effectively be used depends upon the number of processors in the system. Until now, a broadcast mask was assumed to require $2\log_2 N$ bits. Theoretically, only $1.5\log_2 N$ are required. Whenever a bit in the broadcast vector is asserted, the routing bit carries no information. A more efficient way to store the data is to divide each broadcast mask into tuples of three routing bits and three broadcast bits. Each tuple can be represented with five bits; the three routing bits and three broadcast bits together represent 27 states, which is less than 2^5. Using this more efficient method of storing the broadcast masks, an expression for the "break even point" was derived (3.8) The break even point is the number of broadcast masks which can be represented with N+1 bits.

$$BMS = (N+1) / ((5/6)(2)\log_2(N)) \qquad (3.8)$$

BMS is the maximum number of broadcast masks. For a system where N=128, BMS = $129/((5/6)(2)(\log_2(N)) = 11$.

3.8 Grouped Entry Format

Concurrent with the development of our multiple broadcast mask systems, Brooks and Hoag [BrHo90] developed another kind of compromise between the Archibald and Baer entry format and the Censier and Feautrier entry format. Their idea was to group the N processors into G groups each size N/G. At the global table, each entry consists of the two bits of the Archibald and Baer protocol concatenated with G bits. If any of the processors in a group has the block cached, the bit for that group is set.

One can think of this method as a several broadcast mask system where the masks are disjoint, fixed to always represent one set of processors, and together cover all the processors. Continuing with the analogy, each mask has $\log_2(N/G)$ bits set in the broadcast vector of the mask. These asserted bit in the broadcast vector can be any of the $\log_2 N$ bits, but they must be the same for all the masks.

When the number of processors in an entry is small, the processor numbers can be recorded in the bit vector which indicates which groups have the block cached. The interpretation

of the bit vector depends upon the setting of the first two bits of the entry. If these bits are set to Present*, the bit vector should be interpreted as if each bit represented a group, and if the first two bits are set to Present, the bit vector should be interpreted as processor numbers.

The simulator used to simulate the broadcast mask systems was altered to simulate the accuracy of Brooks and Hoag's entry format. For almost all cases, the grouped method was more accurate than the broadcast mask method. The grouped method also does not have the problem of sending out redundant invalidations as the broadcast mask does.

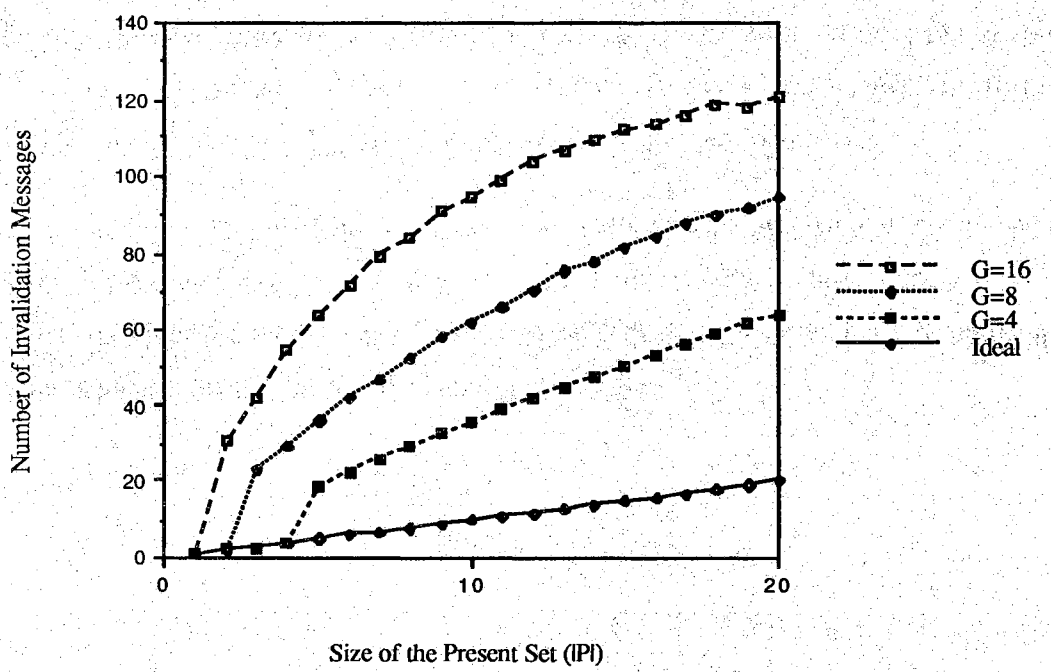
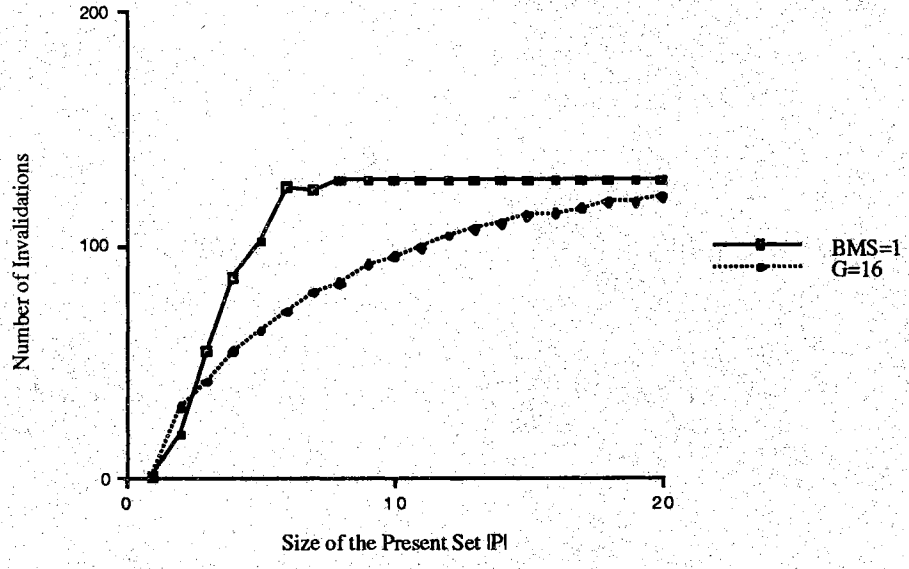


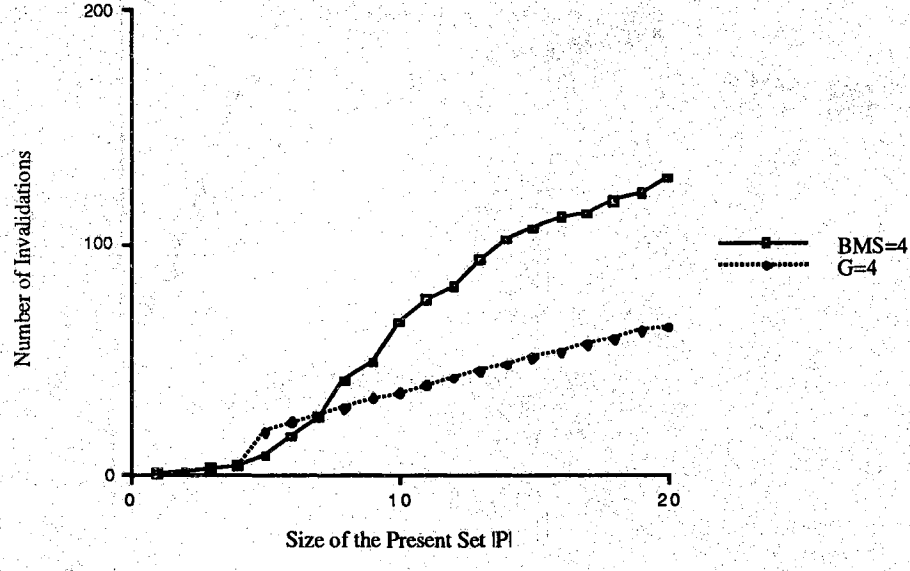
Figure 3.11 Accuracy of Grouped Method (N=128)

3.9 Comparing Accuracy of Grouped Entries to Broadcast Masks

Comparing the accuracy graph of the grouped method to the multiple broadcast mask method shows that only for some cases, when the number of processors is small, the broadcast mask systems are more accurate than the grouped systems.



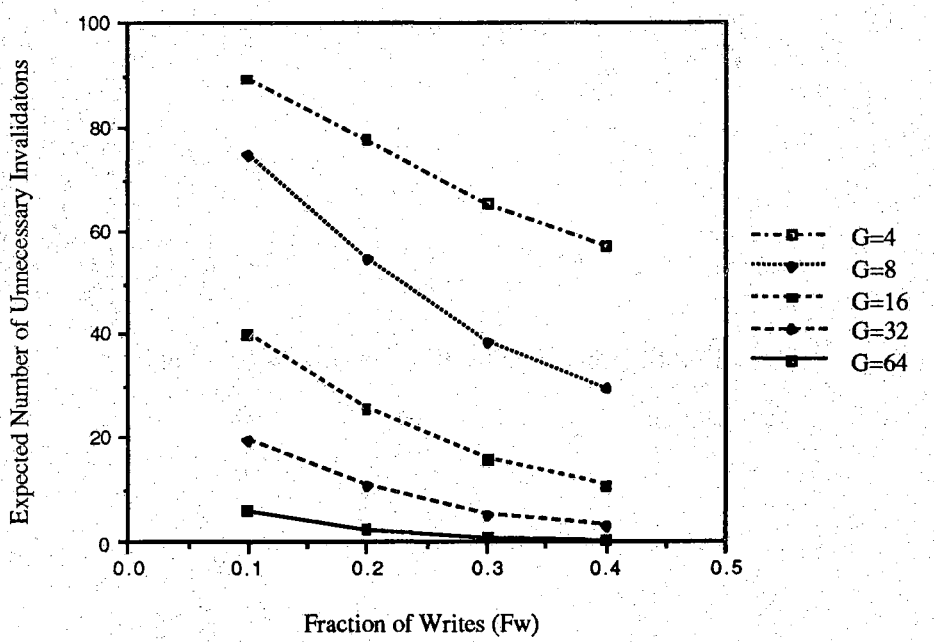
(a)



(b)

Figure 3.12 Broadcast Masks vs. Grouped Method, N=128 (a) G=16, BMS =1 (b) G=4 BMS=4

The expected number of invalidations for the grouped system was determined for various fractions of writes.



Parameters
N = 128
qro = 0.2
qs = 0.2
h = 0.9
Nsb = 512
Ncb = 2048

Figure 3.13 Unnecessary Invalidations for Grouped Method

4. GLOBAL TABLE ORGANIZATIONS

Two classes of global tables will be discussed in this chapter, tables which are copies of the private cache tags first proposed by Tang [Tang76], and tables which have an entry for each block in main memory, proposed by Censier and Feautrier [CeFe78]. The latter class will be discussed first.

4.1 Main Memory Block Recording

For protocols requiring an entry for each block in main memory, the higher order bits of the physical address can serve to select the entry in the global table. The lower order bits of the physical address serve to select which memory module and which byte of the cache block is to be referenced. All operations on global blocks require a global table read, some modification of the entry and a global table write. The modification done by some updating logic depends upon the entry format. Figure 4.1 shows the simplest global table organization for this class of global table.

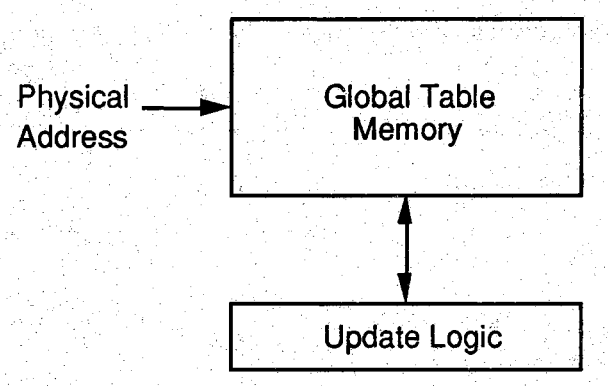


Figure 4.1 Global Table Block Diagram

If the format of the global table entry is an N+1 bit vector, the update logic for the updating of the N bit vector is simply several XOR gates and a multiplexer (Figure 4.2). Besides the logic

shown in Figure 4.2, the update logic must also generate a new RO/RW bit; this is straightforward.

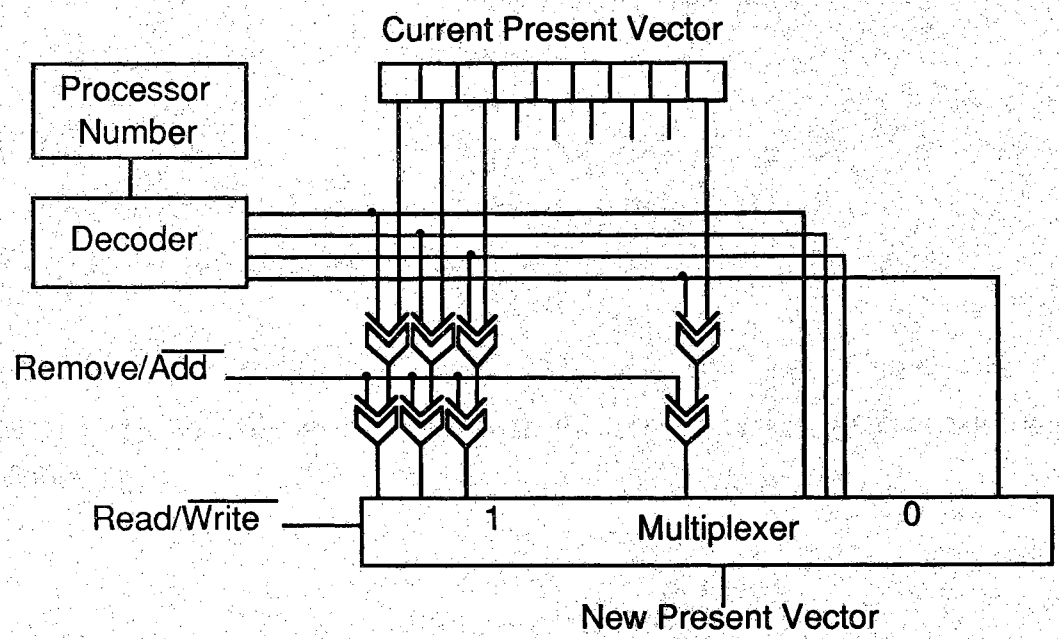


Figure 4.2 Bit Vector Update Logic

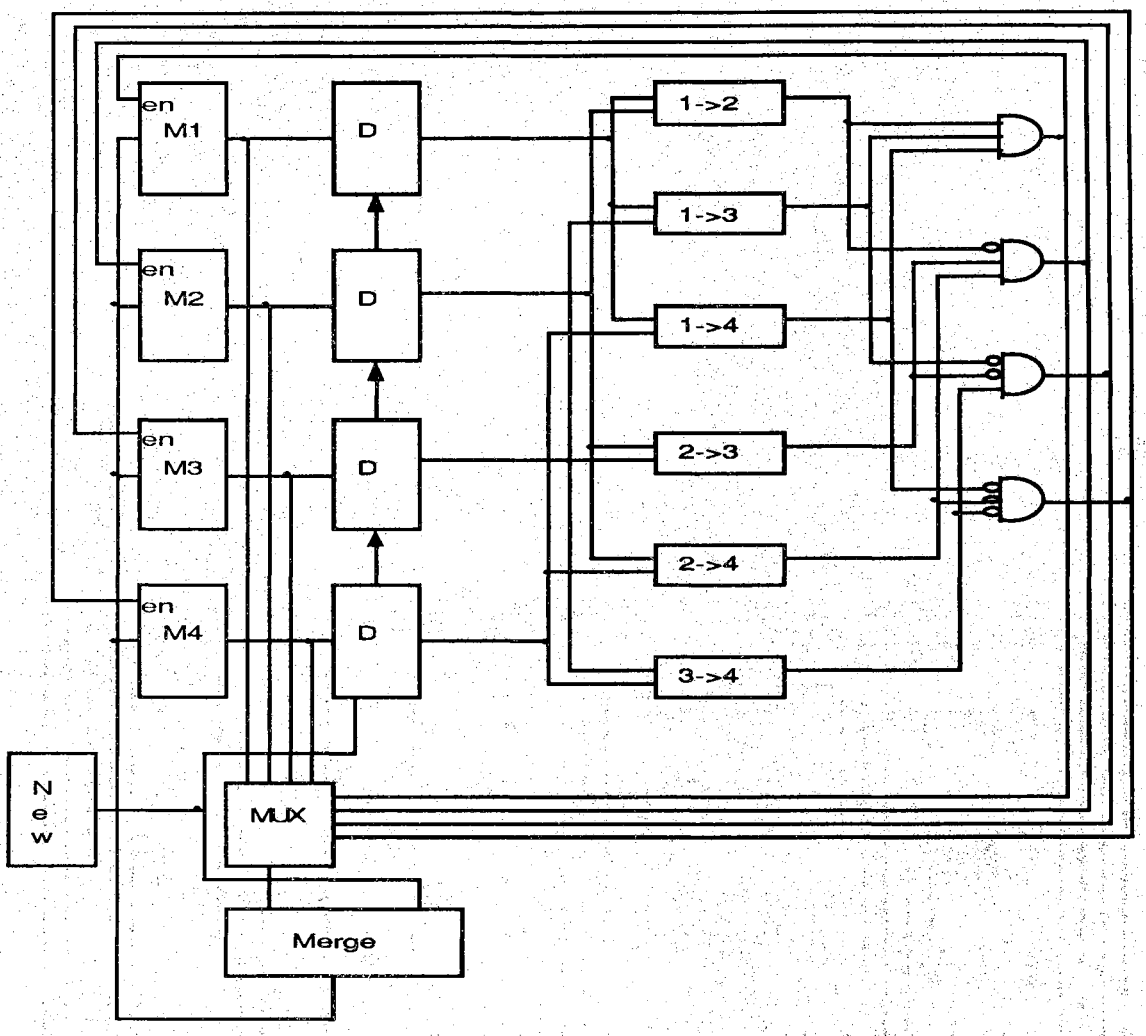
If the entry format is a single broadcast mask, a new processor can be added to the mask if each bit of the broadcast mask is updated according to the logic functions in Figure 4.3. The only cases where a processor can be removed from the broadcast vector is when only one or two processor are in the broadcast mask. The state where RW and B0 are asserted is and unused state, and for the equations below is used to indicate not cached.

```
Bi' = Bi|Ri&Ni
Ri' = Ni&(RO/RW == RW)&B0 | Ri&(not((RO/RW == RW)&B0)

Bi -  Bit i of the broadcast vector
Ri -  Bit i of the routing vector
Ni -  Nit i of the processor number
```

Figure 4.3 Broadcast Mask Interpreting

When multiple broadcast masks and shortest distance merging are used, circuitry must be added to select which of the broadcast mask the new processor will be merged into. To make this selection, each mask computes its distance from new processor number; the closest mask is the one which the processor is merged into. Only (N-1)N/2 comparison circuits are required. Figure 4.4 shows the block diagram of the circuit which selects the closest mask in a four broadcast mask system.



Mi	-	Register holding mask i
D	-	Combinatorial circuit which calculates the distance between a broadcast mask and a processor number
i->j	-	Logic which compares i with j and if i<=j asserts its output
New	-	Register holding the new processor number
Merge	-	Merge circuit defined in Figure 4.3

Figure 4.4 Multiple Broadcast Mask Selector

4.1.1 Pipelining the Global Table

Since the global table performs a read and a write for every memory reference to a shared block, it needs to be fast. One common way to increase performance for any computer systems is to introduce pipelining. The global table and update logic can be thought of as two stations in a

pipeline. A global table reference has a reservation table like Figure 4.5. For operations with this kind of reservation table, the optimal scheduling strategy is the greedy strategy.

	1	2	3
Global Table Memory	X		X
Update Logic		X	

Optimally scheduled operations

	1	2	3	4	5	6	7
Global Table Memory	1	2	1	2	3	4	3
Update Logic		1	2			3	4

Figure 4.5 Pipelined Global Table

When shared memory references are waiting to be serviced, the global table memory is never idle if the table is pipelined, so the performance cannot be increased, and the pipelined global table will service two reference in every four cycles. This is a speed up of 1.5 when compared to the non-pipelined global table, assuming the update logic propagation time equals the global table memory access time.

In order to support pipelining, several latches and data paths must be added to the global table. Figure 4.6 shows where these latches and busses are to be placed.

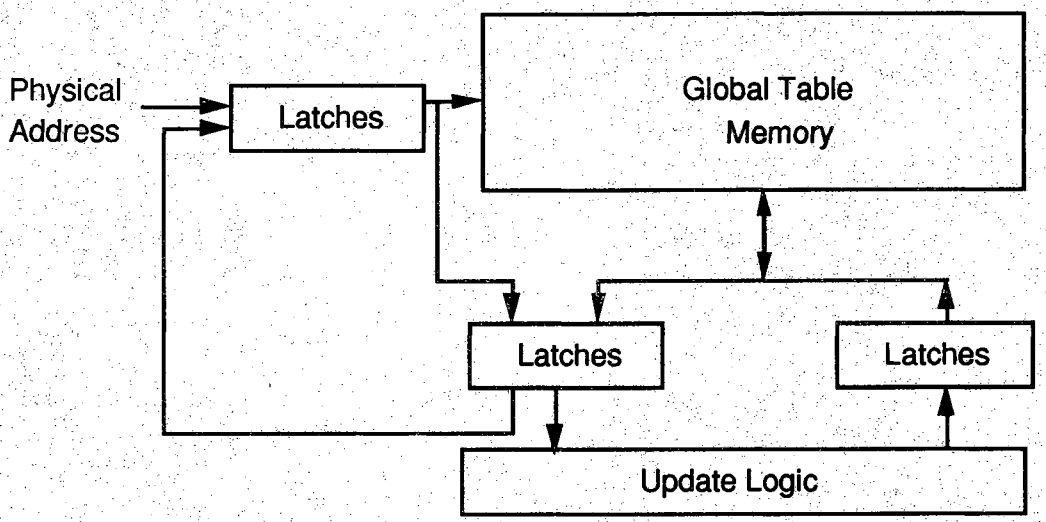


Figure 4.6 Pipelineable Organization

Rather than pipelining to increase throughput, the global table, each global table module can be divided into Nsm sub-modules each serving roughly 1/Nsm of the references. A convenient way to partition the global table is to interleave the module using the lower order bit positions of the physical address (after removing the block offset from the physical address). This increases the throughput of the global table, but does not help when there is contention for one location.

4.1.2 Variable Length Tables

Only those blocks which are cached utilize the many states a "complete" global table entry is capable of representing. At any given time, most blocks are likely to reside in main memory. The motivation behind variable length tables is to reduce the global table size by only maintaining "complete" entries for the blocks which are currently cached. This reduced table of "complete" entries is referred to as the active global table. Because the active global table is smaller than the global table memory, a more exact global table entry format can be used without as much concern for the amount of memory it will use. The physical address cannot be used as a pointer into the active global table, so, for every block in main memory, a pointer into the active global table is kept. This pointer either points to the location in the active global table where the entry for that

block resided or the pointer is NULL, indicating the block resides in main memory. These pointers are stored in the table pointer table.

Figure 4.7 shows a variable length table organization. There are four sections to the variable length table: The free list FIFO, the active global table, the table pointer table, and the update logic. The active global table is the memory which contains the present set information for the blocks which are currently cached. The format for recording the present set information can be any format desired. Although very concise formats, like in Archibald and Baer's protocol, will be poor choices for this type of organization; the table pointer table will be larger than a fixed length global table with this entry format. The free list FIFO maintains which locations in the active global table are invalid. On system start-up, this FIFO must be initialized so it contains all the locations of the active global table.

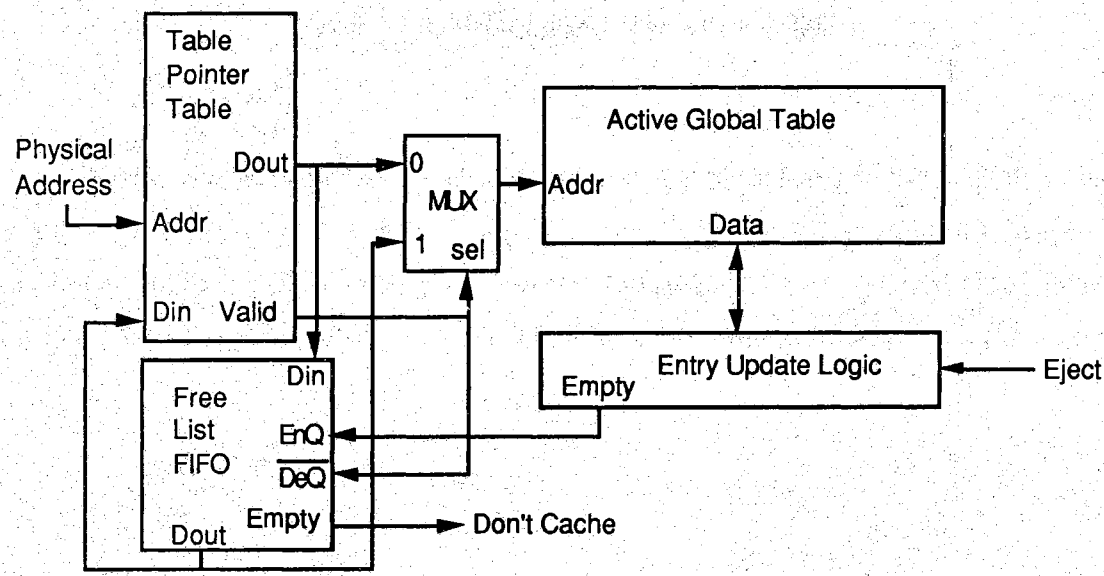


Figure 4.7 Variable Length Table Organization

If every private cache holds unique blocks (a unique block being a block held by no other cache), the maximum number of active global table entries required is (N)(Ncb)/(MM), where Ncb is the number of cache blocks which can be held by a single private cache, and MM is the number of memory modules. This is assuming that all blocks are equally distributed across the memory

modules. The active global table can be made smaller than (N)(Ncb)/(MM) entries, but if the active global table becomes full, indicate by the free list FIFO being empty, the referenced blocks must remain uncached. This allows for an active global table size vs. performance tradeoff. If the active global table size at one global table module is denoted by GS, and the number of bits in each entry is denoted by E, the memory required for the entire global table is:

$$\text{Global Table Size} = (GS*(E+\log_2 GS) + M*\log_2(GS+1))*MM \text{ (bits)} \quad (4.1)$$

With variable length tables stagnant blocks -- blocks ejected from all private caches but not removed form the global table -- can fill the active global table. To avoid this problem tidy ejection must be employed. When the last processor to have a block cached ejects it, the entry is relinquished by enqueueing the pointer to the entry onto the free list FIFO. In order to use broadcast masks as the entry in the active global table, a count field must be appended to each entry to indicate the number of processors which have the block. Whenever a block ejection reaches the global table the counter is decremented. If an ejection operation occurs and the value of the counter is one, the active global table location can be given up.

Strenstrom proposed the present vector be migrated into the private caches [Stre89], and obtaining similar reduction in the global table size. However, when his protocol is used, and a processor ejects a block from its cache (and it is the owner), finding another processor to take over ownership requires O(N) messages to and from other processors. Furthermore, the cache controller unit is likely to be very complex. Variable length tables offer comparable size reduction, size flexibility, and reduced network traffic.

4.1.2.1 Pipelining Variable Length Tables

As with fixed length tables, pipelining can be used to increase the throughput of the system. The four basic stations of the variable length table are the table pointer table, the active global table, the update logic, and the free list FIFO. When and what stations an operation requires depends upon the type of operation (e.g. read miss, ejection etc.) and what state the block is currently in. The reservation stations for all possible global table operations are shown in Figure 4.8.

	1	2	3	4	
Ejection					Group 1
Table Pointer Table	X				
Active Global Table		X		X	
Update Logic			X		
Free List FIFO				X	

	1	2	3		
Read Miss (Not Cached) or Write Miss (Not Cached)					Group 2
Table Pointer Table	X		X		
Active Global Table		X			
Update Logic		X			
Free List FIFO		X			

	1	2	3	4	
Read Miss (RO or RW), Write Miss (RO or RW), Write Hit (RO), or Flush					Group 3
Table Pointer Table	X				
Active Global Table		X		X	
Update Logic			X		
Free List FIFO					

Figure 4.8 Reservation Tables for Variable Length Table

Since the state of the cache block is not known until after the table pointer table and the active global table has been accessed, one cannot use the reservation tables in Figure 4.8 to develop a greedy pipeline scheduler [Kogg81]. Greedy scheduling can realistically be implemented in hardware [Davi71]. The collision vectors used by a greedy scheduler must not depend upon the state of the cache block being referenced. Reservation tables which only depend on the type of operation are shown in Figure 4.9.

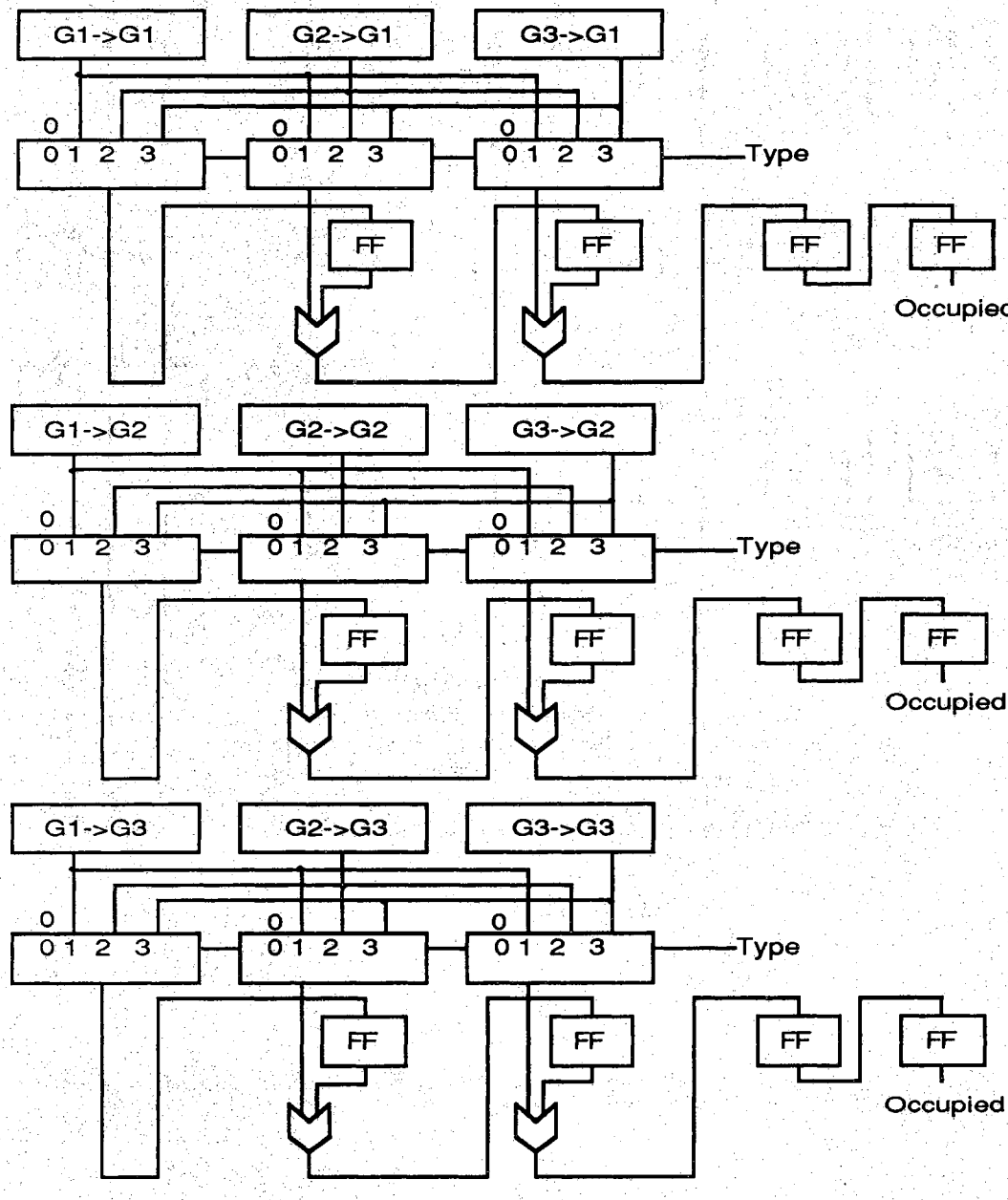
	1	2	3	4	
Ejection					Group 1
Table Pointer Table	X				
Active Global Table		X		X	
Update Logic			X		
Free List FIFO				X	

	1	2	3	4	
Read Miss or Write Miss					Group 2
Table Pointer Table	X		X		
Active Global Table		X	X	X	
Update Logic		X	X		
Free List FIFO		X			

	1	2	3	4	
Write Hit or Flush					Group 3
Table Pointer Table	X				
Active Global Table		X		X	
Update Logic			X		
Free List FIFO					

Figure 4.9 Reservation Tables for Realistic Variable Length Table

A greedy pipeline scheduler can be built for a global table using three shift registers because there are only three unique reservation tables. Each of the shift registers determines when a new operation of the corresponding type can be initiated. When a new operation is scheduled, all the shift registers are updated by ORing in the appropriate collision vector. Figure 4.10 shows the circuit.



Type	Action
0	No operation initiated
1	A group 1 operation was initiated
2	A group 2 operation was initiated
3	A group 3 operation was initiated
Gi-> Gj	Collision vector for a group i operation initiated before a group j operation

Figure 4.10 Greedy Scheduler

When an operation belonging to group i wants to perform a global table operation, the last bit of the ith shift register is checked. If this bit is asserted, it must wait until it is cleared before initiating the operation. When the operation is initiated, whenever that might be, it must place its group number, i, onto the type lines so all the shift registers become aware of the operation in progress.

Collision vectors which could be used in the greedy scheduler are shown in Figure 4.11.

2nd/1st	G1	G2	G3
G1	010	110	010
G2	110	110	110
G3	010	110	010

Figure 4.11 Collision Vector Table

4.1.2.2 Simulating the Variable Length Table

A simulator was made which generates operations randomly and schedules them using the greedy strategy. The operation generation subroutine of the simulator generates arrival times as a Poisson process. The arrival times where made to be a Poisson process for several reasons. In general, Poisson processes are good models for traffic because a Poisson process is the only one-at-a-time random process which has stationary and independent increments. Stationary increments implies that the rate of arrival is constant for all time, and independent increments implies that the number of operations in one time interval does not influence the number of time arrivals in a disjoint time interval. For a Poisson process, the E[#number of arrival in t time units] = λt. Fittingly, λ is called the rate of the process, and $1/\lambda$ is the expected time between arrivals. Though memory traffic is usually bursty, modelling the arrivals as a Poisson process for a range of arrival rates indicates how the table will perform at busy and as well as slow times.

The interarrival times of the Poisson process (with rate λ) are distributed as independent and identically distributed exponential random variables with parameter λ. In order to generate these random variables on the computer a function which maps the uniform random variable, U, ranging from 0 to 1 (available of the system) to an exponential random variable was derived.

Interarrival Time = f(U) (4.2)

Using probability theory, f was found.

$$f(U) = \frac{-1}{\lambda}\ln(U) \qquad (4.3)$$

An "ideally" pipelined greedy system was also simulated using the greedy algorithm, even though it in actuality cannot be implemented. This gives an indication of what kind of performance is lost by the using collision vectors generated from the reservation table in Figure 4.9 rather than from the reservation table in Figure 4.8. The "ideal" scheduler uses the collision vectors in Figure 4.12, which are generated from the reservation table in Figure 4.8.

2nd/1st	G1	G2	G3
G1	010	11	010
G2	110	01	100
G3	010	11	010

Figure 4.12 Collision Vector Table for Ideal Scheduling

The average number of cycles an operation has to wait depends upon the rate at which the references to the global table arrive. Figure 4.13 shows the average number of cycles an operation has to wait before being serviced for the non-pipelined case, the pipelined case, and the ideally pipelined case. This is for a snapshot of nine thousand global table operations. (Note: the y-axis of the graph is logarithmic)

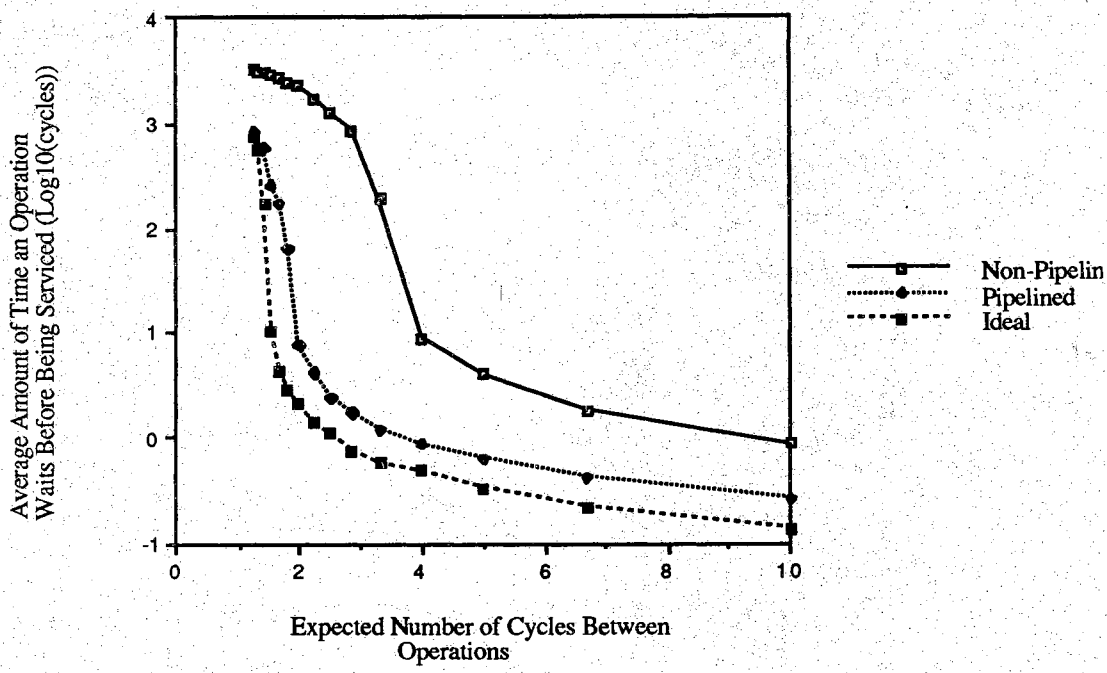


Figure 4.13 Average Waiting Time

The simulation which generated the result presented in Figure 4.13 simulated nine thousand operations where 30% of the operations were group 1 operations, 45% were group 2 operations, and 25% were group 3 operations. For the "ideal" pipeline, the distribution was 30% group 1, 10% group 2, and 60% group 3. The reason the group distributions are different is because the basic operations are put into different groups as shown in Figures 4.9 and 4.10. The percentages were chosen so to make the comparison fair.

Notice how each configuration appears to have a maximum throughput which the global table can handle. If the rate at which operation arrives exceeds this threshold, the waiting time drastically increases. This is because the the operations later in the simulation have to wait a significant portion of time. In a real system, global table operations will not wait this long. The limited throughput of the global table coupled with heavy traffic will cause the buffers at the memory modules to fill, stopping the processors from sending any more references, or a processor

may have to wait because of data dependencies. In both cases, system performance will drop because of the global table.

The speedup achieved by pipelining the global table was estimated by comparing the time it took to service nine thousand global table operations in the pipelined, "ideally" pipelined, and non-pipelined cases. The service time of nine thousand operations was simulated at various arrival rates. Figure 4.14 shows the speedup for the realistic and ideal system.

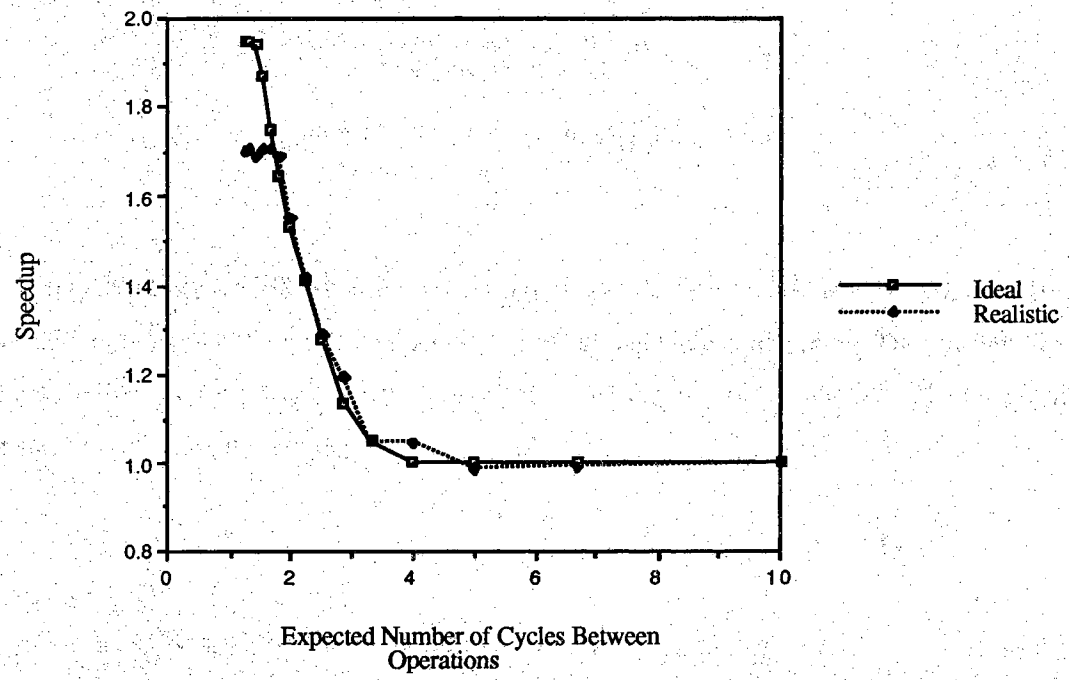
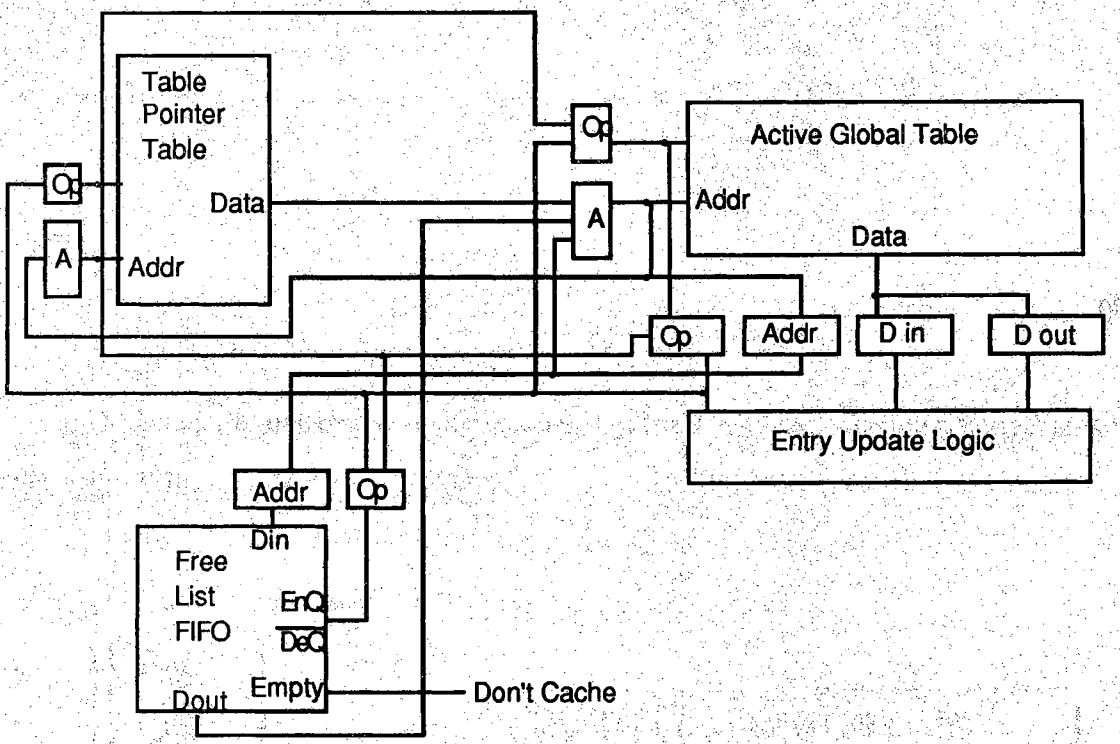


Figure 4.14 Speedup Due to Pipelining Table

Speedup only occurs when the arrival rate becomes sufficiently quick. When traffic is sparse, seldom is there more than one operation at the global table at one time. The speedup levels off at approximately 1.7 for the realist pipeline, and 1.95 for the "ideal" pipeline. These speedup limits are the maximum throughput rates of the various tables.

Redesigning the global table so that it can be pipelined involves including staging registers and extra data paths as shown in Figure 4.15.



A or Addr - Address Latch
D - Data Latch
Op - "Operation type" latches and control logic

Figure 4.15 Pipelined Variable Length Global Table

As with fixed length tables, the global table module can be divide into Nsm submodules each handling 1/Nsm of the locations. Ideally, this will distribute the traffic so each submodule handles 1/Nsm of the traffic. From the modelling point of view, if the trial to decide which submodule should receive the operation is independent of when the operations occurs, the arrivals of operations observed by each of the submodules is a Poisson process as well, but this time with a rate of λ/Nsm. Because of this property of the Poisson process the graphs in Figure 4.13 and 4.14 represent the analysis for systems with submodules as well.

4.2 Private Cache Tag Entry Tables

The other class of global table which will be discussed holds copies of the private cache tags rather than holding an entry for each block in main memory. This kind of global table is required by Tang's [Tang76] protocol. The amount of memory required for this type of table is :

$$\begin{aligned} \text{Table Size} &= \text{Ncb*N*(Number of bits in a cache tag) (bits)} \\ &= \text{Ncb*N*}\log_2(M)\ \text{(bits)} \\ &< \text{N*M if } \log_2(M*Ncb) < M \end{aligned} \qquad (4.4)$$

If $\log_2(M)$*Ncb is much less than M, this alternative organization will produce a smaller global table then a fixed length global table with an N+1 bit entry format.

For global table operations, the tag field of the address must be compared with all copies of the private cache tags from all processors which map to the same set. A match indicates that the block is cached. A bit vector can be constructed representing the present set if all the match bits are concatenated together as shown in Figure 4.16.

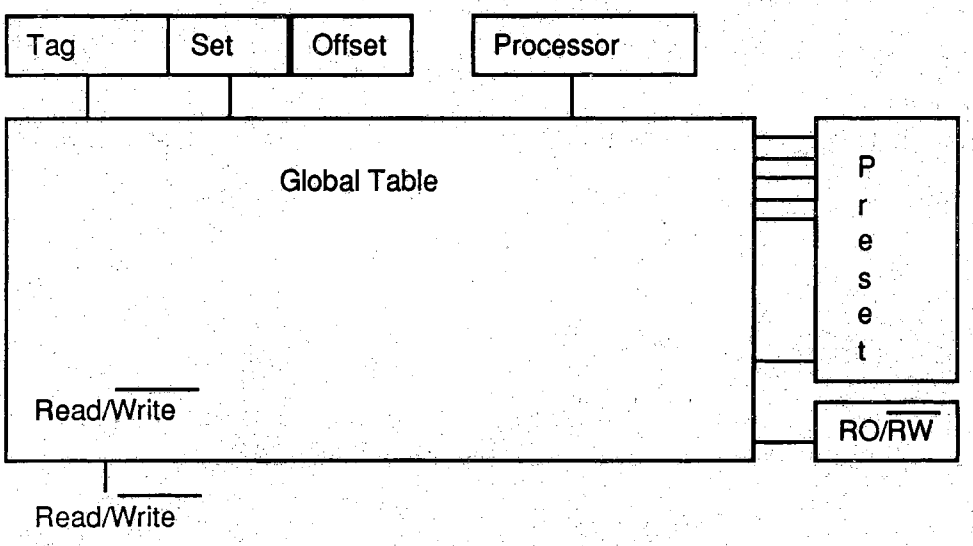


Figure 4.16 Block Diagram of Tang's Global Table

One way to build the global table is with Ns sets of N*K way associative memory where each word is a private cache tag. (Figure 4.17 shows the organization of one set) As mentioned

earlier, the major problem with this approach is the associativity. For the case of CMOS, the access time of the associative memory is dominated by the capacitance of on the lines that the physical address must be placed on. This is not the only place where there might be a high capacitive loading. If busses are used for distributing the physical address and capturing the present vector, the access of the table is likely to be made even slower. These busses can be replaces with a large decoder and multiplexer constructed with logic gates. The disadvantage here is the area the multiplexer and demultiplexer takes up on a chip.

5. SYNCHRONIZATION VARIABLES

In order to implement critical sections, a multiprocessor must have an uninterruptable read-modify-write operation like a test-and-set [PeSi85]. One solution is to reserve several locations of the main memory to act as this type of variables, and have special entry formats and hardware for these variables. A flexible way to implement the uninterruptability of operations with synchronization variables is to allow these locations to be locked. A locked variable has the unique property that it can only be accessed by one processor at a time. A processor referencing a variable which is locked by another processor must wait until the processor which has control of the variable unlocks it. To perform an uninterruptable read-modify-write operation, a processor locks the variable, reads it, performs any modification it wants to, then unlocks the variable.

If a processor has control of a variable, and other processors are trying to lock the variable, the global table might send retry messages to these processors which don't have the variable. Since the waiting processors don't know when the variable will become unlocked, they have to continually re-request to lock the variable. This is what is called busy waiting, and it is highly undesirable because accesses to synchronization variables are frequent, and network traffic effects the system performance.

Adaptive back-off techniques have been proposed by Agarwal and Cherian [AgCh89]. The idea behind these back-off techniques is to delay re-requesting the variable by some amount of time. The amount of time usually depends upon the number of retry messages received. These techniques still require re-request messages to be sent, but have better performance than busy waiting.

5.1 An Economical Queuing Entry Format

A better way to handle access to synchronization variables is to record which processors have requested the variable, and when the variable is unlocked, pass control to one of the waiting

processors. To inform the processor of the unlocking, a message must be sent to the new processor which controls the variable. This way, requesting processors don't need to request for a lock more than once. Techniques for queuing requests to synchronization variables have been proposed by Gottlieb et. al. [GoLR83] and altered by Goodman et. al. [GoVM89] so as to not require a combining network. Both these techniques require a front pointer, a rear pointer, and a buffer to store waiting processor numbers. For large scale multiprocessors, the queue length might need to be limited, in which case retry messages must be generated when the buffer fills.

An easier way to record the requestors, than using circular waiting queues which record processor numbers, is to simply record which processors are waiting in a bit vector, and grant the variable to the waiting processor with the lowest number. An example entry format is shown in Figure 5.1.

Figure 5.1 Synchronization Variable Global Table Entry Format

The L bit in Figure 5.1 indicates if the variable is currently locked. If L is asserted, the Controller field indicates the processor number of the processor in control of the variable. The Waiting Vector is the N bit vector indicating which processors are waiting to control the block. When a lock request is received by the memory module, and the variable is locked, the processor number is recorded by asserting the corresponding bit in the Waiting Vector. The global table then sends a response back to the processor indicating it should wait. When the block is granted to the processor the global table module sends a granting message to the processor. Figure 5.2 shows this pictorially.

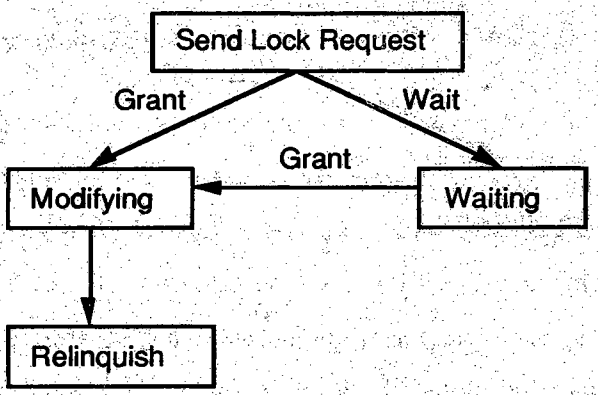


Figure 5.2 Synchronization Variable Access Diagram

While in the waiting state, the processor could simply wait, spinning on a No-Op in its cache, or it could execute some other process until the variable is granted.

Starvation is possible if the synchronization is handled in the way just described. Imagine a sixteen processor system where processors four, six, and ten are contending for a particular synchronization variable. If processor four gets control of the variable first, and processors six and ten request, then six and ten will receive wait responses because the variable is already locked. Processors six and ten will be recorded in the waiting vector, and when processor four relinquishes the variable, processor six will be granted control. If, while processor six is modifying the variable, processor four re-requests, processor four will be added to the waiting vector. When processor six relinquishes the variable, processor four will regain control, because four is less than ten. This cycle can continue indefinitely, starving processor ten. A way to avoid the starvation problem is to decompose the waiting vector into two N bit vectors: Priority 1 Vector and Priority 2 Vector.

Figure 5.3 Non-Starving Synchronization Variable Entry Format

When a new processor requests a synchronization variable, it is added into Priority 1 Vector if its processor number is greater than the controller's processor number, otherwise it is placed in Priority 2 Vector. So, for the example described above, processor four will be placed into Priority 2 Vector the second time it requests -- allowing processor ten to access the variable.

To reduce the amount of network traffic, two compound operations can be implemented: Lock&Fetch and Store&Unlock. With these compound operations a processor can increment a shared counter with just three network messages. Lock&Fetch will automatically send the variable to the processor with the grant lock message, and Store&Unlock will store a value to the variable before it unlocks it. The algorithm which the global table must follow for the Lock&Fetch and Store&Unlock Operations is presented in Figure 5.4. The algorithm assumes that the current operation was initiated by processor i and references synchronization variable s. The entry format in Figure 5.3 is analogous to the data structure in 5.4 (a).

(a)

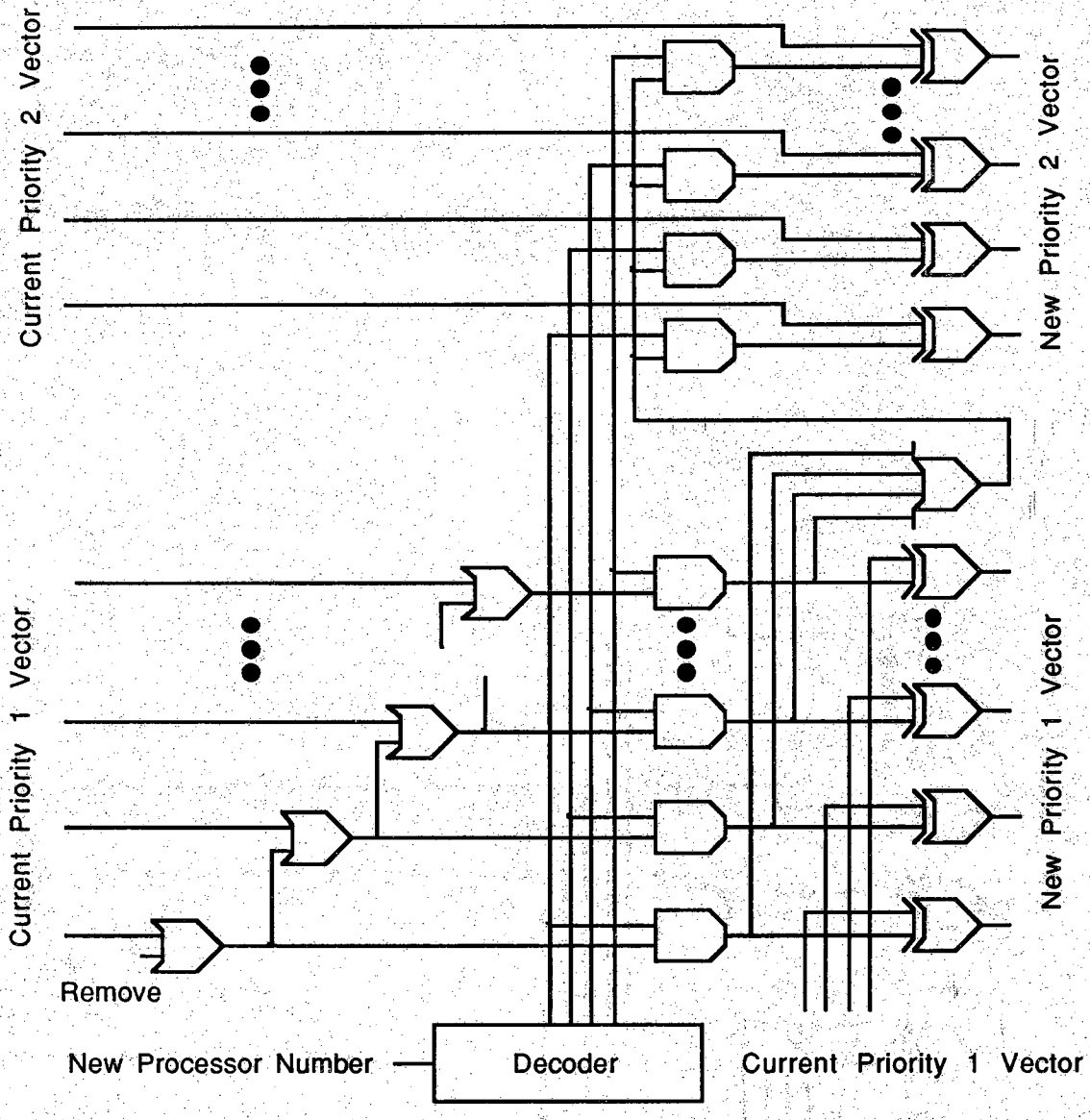
```
type
   synch_var_type =  record
                       locked:boolean;
                       controller:1..N;
                       priority2,priority1:array[1..N] of boolean;
                     end;
var
   table[0..number of synchronization variables] of synch_var_type;
```

(b)

```
switch (operation.type) of
   case lock&fetch:
      if(table[s].locked = false) begin
         table[s].controller = i;
         table[s].locked = true;
         send granted message to i with contents of s
      end
      else  begin
         if(table[s].controller < i)
            table[s].priority1[i] = true;
         else
            table[s].priority2[i] = true;
         send wait message to i
      end
   case store&unlock:
      s = new value;
      if(for every x 0<x≤N table[s].priority1[x] = false) begin
         if(for every x 0<x≤N table[s].priority2[x] = false)
            table[s].locked = false;
         else  begin
            j = smallest j such that table[s].priority2[j] = true;
            table[s].controller = j;
            table[s].priority2[j] = false;
            send granted message to j with contents of s
         end
      else  begin
         j = smallest j such that table[s].priority1[j] = true;
         table[s].controller = j;
         table[s].priority1[j] = false;
         send granted message to j with contents of s
      end
endswitch
```

Figure 5.4 Lock&Fetch and Store&Unlock Algorithm (a) Entry Format (b) Algorithm

The circuit for updating Priority 1 Vector and the Priority 2 Vector of a synchronization variable format is shown in Figure 5.5. Notice this is not a sequential circuit; the updating can be done with combinational logic.



Remove - Signal asserted if the the synchronization variable is being unlocked

Figure 5.5 Synchronization Entry Updating Logic

5.2 Evaluation of the Lock Granting Algorithm

The Lock&Fetch and Store&Unlock algorithms were assessed by writing a simulation program. The simulation models one synchronization variable trying to be accessed by several

processors. Each processor, for the simulator, is in one of three states: working, waiting, or locking. When in the working state, processors are performing useful computations. When a processor is in the waiting state, it is waiting to access the synchronization variable, and the locking state represents when a processor has control of the synchronization variable. The time between when a processor unlocks the variable until it re-request it, is assumed to be exponential with parameter λ. As lambda decreases, the expected time increases, because the expected value of an exponential random variable is $1/\lambda$. The reasons for choosing an exponential waiting time are the same as they were for the interarrival times of global table operations in the previous chapter.

The simulator recorded the time each processor spends in the waiting state before each time it controls the synchronization variable, and the number of times each processor controls the variable. It does this for both two priority and first in first out (FIFO) arbitration. Figure 5.6 shows the waiting times for various requesting rates. The parenthetic numbers in the figure key indicate the expected number of cycles spent in the working state between requests.

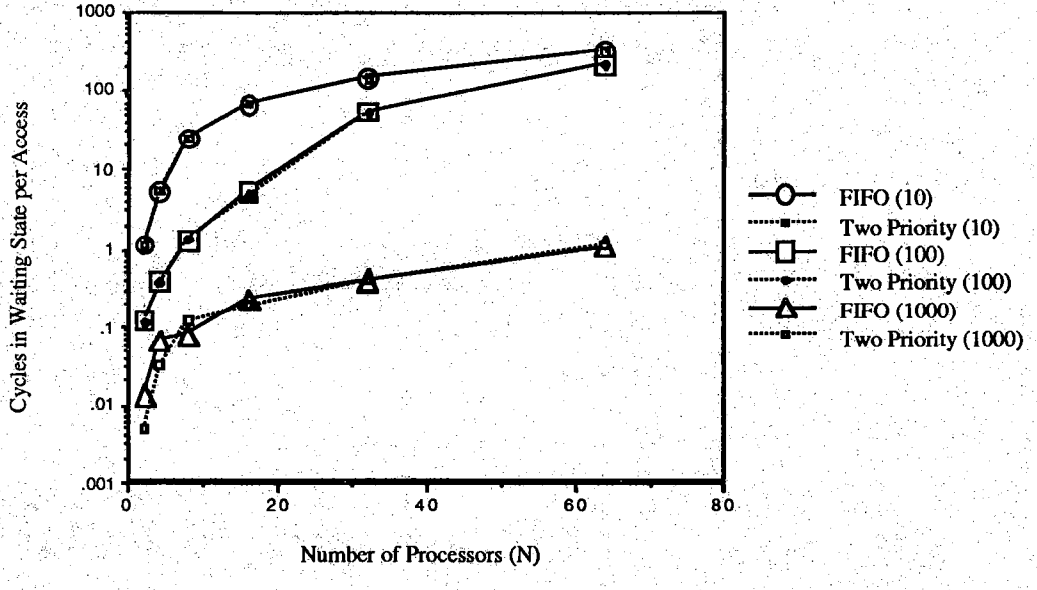


Figure 5.6 Waiting Time per Access

Because the granting algorithm chooses processors based upon processor number, there is concern that this scheme might not be fair, however, even when sixty four processors are trying to access the synchronization variable, a situation where processor favoritism would be noticeable if it existed, the two priority scheme appeared to be just as fair as first in first out. An interval of time in which 6400 references to the synchronization variable were made is shown in Figure 5.7

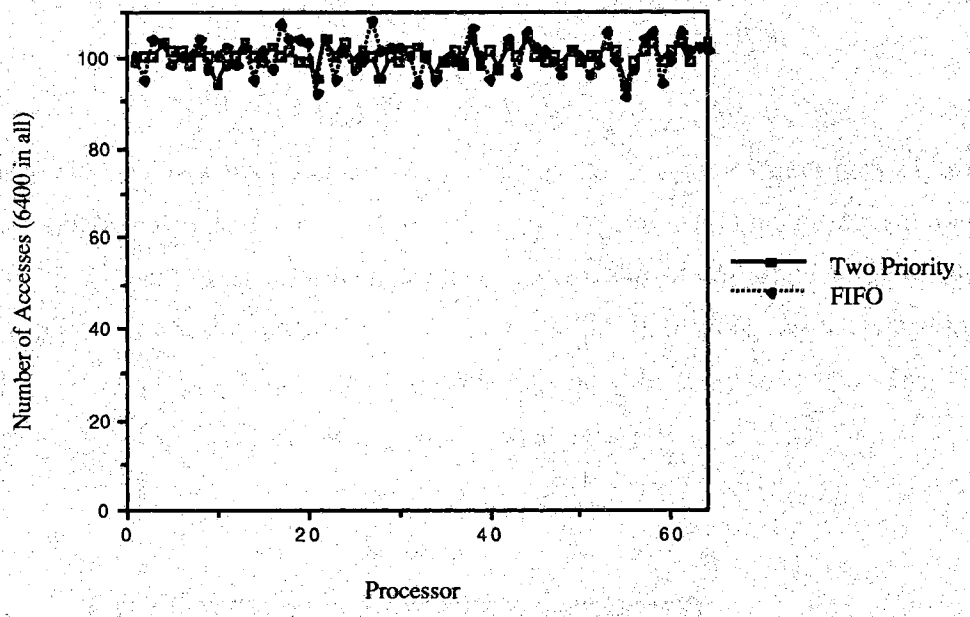


Figure 5.7 Arbitration Fairness

6. LINKED LIST SYSTEMS

As discussed in chapter three, the amount of network traffic introduced in the two counter system is considerable (see figure 6.1). Many of these messages are return receipts.

Operation	Number of Messages
Read Miss	
Not Cached	2
RO	2
RW	4
Write Miss	
Not Cached	2
Cached RO	2|P| + 2
Cached RW	3
Write Hit	
Cached RO	2(|P| -1) + 2

Figure 6.1 Number of Messages in Two Counter Systems

Some variations of linked list protocols require fewer messages to inform the writing processor that the write is performed with respect to all processors, reducing network traffic. This is the primary motive for considering linked list protocols. Furthermore, linked list protocols require a modest amount of memory to construct the global table.

6.1 Singly Linked List Protocol

Linked list directory protocols maintain coherency by forming a linked list of processors which have the block cached. Thaper and Delagi [DeTh90] presented an outline for what will be referred to as singly linked list protocols; this is the only publication on this topic.

With linked list protocols, there still exists a global table distributed across all the memory modules. Each block in memory has a global table entry; the entry points to the head of the list of processor which have the block cached. If this pointer is NULL, the block is assumed to reside in

the memory module. Stored with the pointer to the head of the list is a RO/RW bit which indicates whether or not the processor pointed to by the entry has RW privilege to the block.

Associated with each block in every private cache is a pointer to the next processor in the list. Figure 6.2 is a Pascal-like type definition which is analogous to the memory required for the global table.

```
type
   processor_number_type = 1..N
   tag_type = 1..MAXTAG {MAXTAG is the maximum cache tag value}
   entry_type = record
                  cached:boolean;
                  rw:boolean;
                  head:processor_number_type;
                end
   cache_entry_type = record
                         dirty:boolean;
                         null:boolean;
                         next:processor_number_type;
                         tag:tag_type;
                      end
var
   table:array[1..M] of entry_type;
   cachetable: array[1..N] [1..Ncb] of cache_entry_type;
```

Figure 6.2 Software Analogy to Table

The variable `table` is analogous to the table distributed across the memory modules, and `cachetable` is analogous to the pointers stored with the cache tags of the private caches.

Consider, for example, when processors three, five, and thirtyone have block seven cached. One way this present set information could be recorded is shown below.

```
table[7] = {cached=true; rw=false; head=5}
cachetable[3][7] = {null=true}
cachetable[5][7] = {null=false; next=31}
cachetable[31][7] = {null=false; next=3}
Global Table->processor5->processor31->processor3
```

Figure 6.3 An Example Block

All |P|! permutations of the processors which have the block cached are correct representations of the present set (the order of this list depends on when the processors access the block for the first time)

The total amount of memory required by a singly linked list protocol is:

$$\text{Total memory} = M*(\log_2 N+2) + Ncb*N(\log_2 N+1) \text{ bits} \quad (6.1)$$

Provided Ncb*N is on the same order as M, the amount of memory required is comparable to a single broadcast mask system.

6.2 Specification of Singly Linked List Protocol

A description of the protocol is inherent in the description of how the system reacts to the five basic operations of the private caches: read hit, read miss, write hit, write miss, and ejection. The actions which are performed in response to operations depend upon the current state of the linked list, except in the case of a read hit. The protocol described below assumes processor i is acting on block b.

Read Hit : (`cachetable[i][b mod Ncb].tag` matches address)

The word is read from the private cache.

Read Miss:

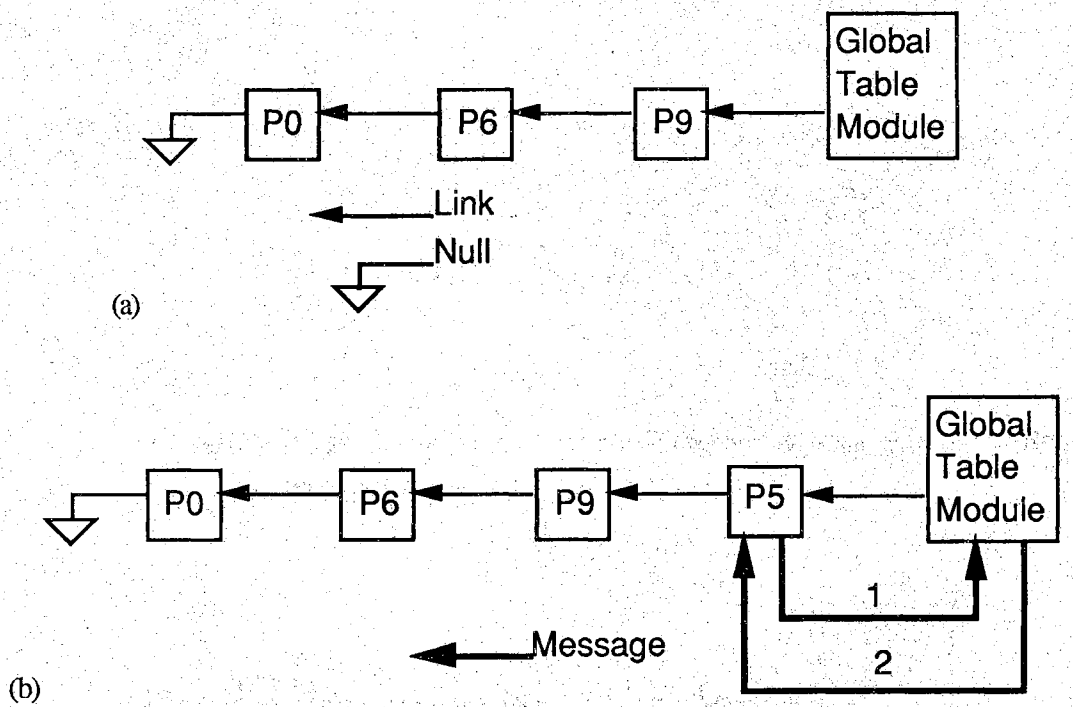
Not Cached: (`table[b].cached = false`)

A read request is sent to the memory module responsible for b. The block is fetched, and `table[b].head` is updated to point to processor i. `Table[b].cached` is asserted; `table[b].rw` is deasserted.

Shared RO: (`table[b].cached=true and table[b].rw=false`)

A read request is sent to the memory module responsible for b. The block is fetched, and `table[b].head` is updated to point to processor i. The old `table[b].head` value is passed to processor i, along with the block, and stored in `cachetable[i][b mod Ncb].next`. This effectively adds processor i to the front of the list of processors which have the block cached. Figure 6.4 shows

the system before and after processor five has a read miss on a block already cached by processors nine six, and zero.



Explanation of Messages
1) A read miss message is sent to the global table module.
2) The block and the pointer to processor nine are sent to processor five.

Figure 6.4 Read Miss by Processor Five (a) Before (b) After

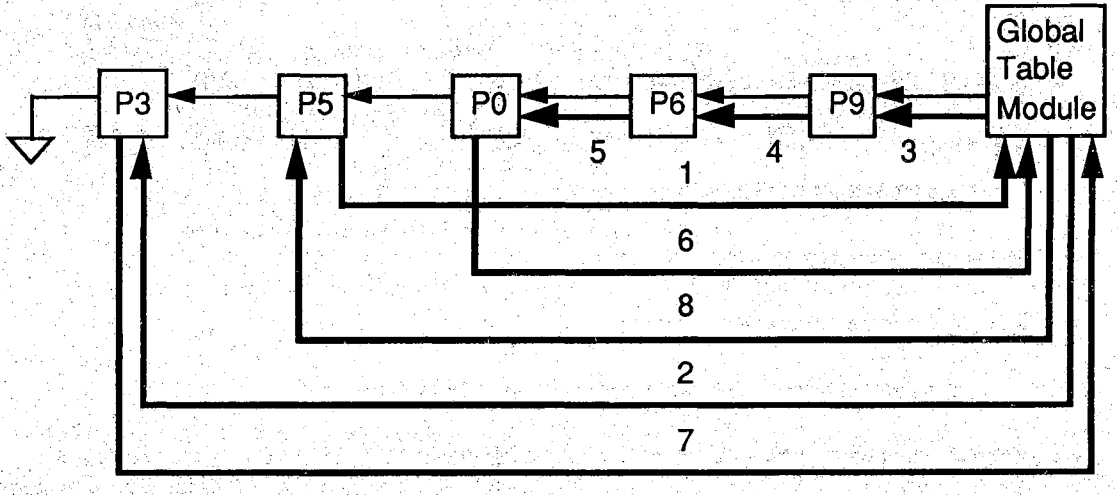
RW: `(table[b].cached = true and table[b].rw = true)`

A read request is sent to the memory module responsible for block b. Upon realizing that the block is cached RW, the memory module sends a flush message to the processor which has the block. When the dirty block arrives at the memory module, it is relayed to processor i along with the old `table[b].head` pointer. `Table[b].head` is set to the requesting processor number, and `cachetable[i].next` is set to the old value of `table[b].head`. The block is then set to RO status by clearing table[b].rw.

Write Hit:

Shared RO: `(table[b].cached=true and table[b].rw=false)`

A write hit message is sent to the memory module responsible for block b. Along with write request, the processor number (`cachetable[i][b mod Ncb].next`) is sent. Processor i, immediately after the write hit message is sent, starts executing its next memory reference. Processor i must send its next pointer to the memory module, and the memory module can initiate an invalidation message to `cachetable[i][b mod Ncb].next` only after it realized there are two streams of invalidation -- one from the memory module to the processor i, and one from processors i to the end of the list. The memory module must initiate the invalidation of the processors at the end of the list because there must be an exact way to tell when a write is performed with respect to all processors. Figure 6.5 is an example showing processor five performing a write to a clean block which it has cached; in the example, processors nine, six, zero, and three also have the block cached.



Explanation of Messages

1) Processor five sends the write hit message to the memory module.
2) Processor `cache[i][b mod Ncb].next` receives an invalidation message.
3-5) The list is walked, invalidating along the way.
6) Processor zero sends a message to the memory module indicating the stream of invalidations is complete.
7) Processor three sends a message to the memory module indicating the stream of invalidations is complete.
8) The memory module sends a message to processor five, telling it that the write is performed with respect to all processors.

Figure 6.5 Write Hit by Processor Five to a Shared Block

At the global table module, `table[b].head` is assigned to i, and `table[b].rw is set.`

When a processor, j, receives an invalidation message, it invalidates the block it is instructed to, then generates a invalidation message to `cachetable[j][b mod Ncb].next`. No invalidation message is generated if `cachetable[j][b mod Ncb].next` is i (the processor ejecting the block) or if `cachetable[j][b mod Ncb].valid` is false.

RW: `(cachetable[i][b mod Ncb].dirty = true)`

It is known that processor i is the only processor with RW privilege by the fact that `cachetable[i][b].dirty` is asserted, so the cache may be written to without updating main memory.

Write Miss:

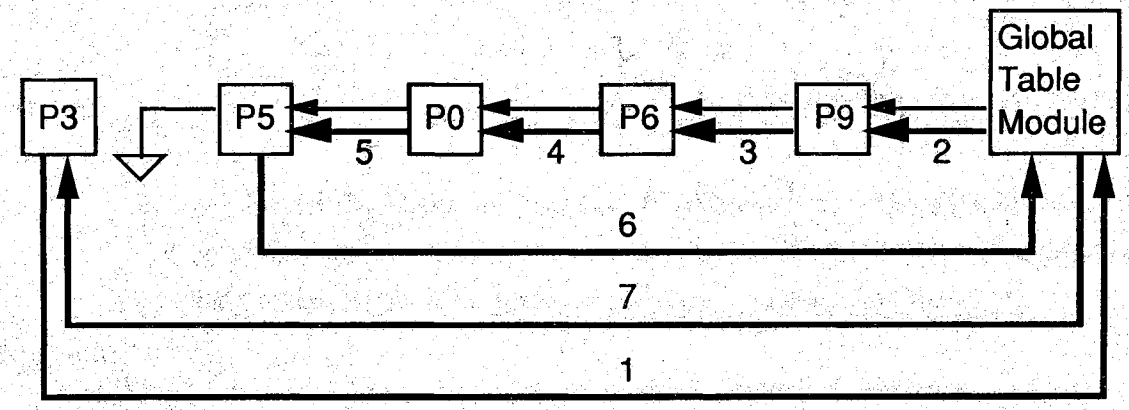
In the event of a write miss, a message is sent to the memory module which corresponds with the address. When that message arrives at the memory, `table[b]` is looked up.

Not Cached: `(table[b].cached = false)`

`Table[b].head` is assigned to i and `table[b].rw` is asserted. While the global table is being changed, the block can be fetched. Once fetched, the block is sent to processor i.

Shared RO: `(table[b].cached = true, table[b].rw = false)`

The processors which have the block must be invalidated, and the linked list modified so that only processor i is in the list. An invalidation sequence is initiated by sending a message to processor `table[b].head`. The linked list of processors is walked, and one by one they are invalidated. While the invalidations are occurring, the block can be fetched from main memory and sent to processor i. `Table[b].first` is assigned to i and `table[b].rw` is asserted.



1) A write miss message is sent to the memory module
2-5) The list is walked, invalidating each processor's copy of the block.
6) Processor five sends a message indicates the list is invalidated.
7) Processor three is notified that the write is completed.

Figure 6.6 Write Miss by Processor Three to a Shared Block

RW: `(table[b].cached = true, table[b].rw = true)`

An invalidation message must be sent to processor table[b].head. Once the block has been sent back to memory, the block is sent to the requesting processor and `table[b].head` is set to i; `table[b].rw` is also asserted.

Ejection:

RW: `(cachetable[i][b mod Ncb].dirty = true)`

A block which is cached with RW privilege is not coherent with respect to memory, so when ejected the block must be sent back to memory, and the global table must be updated by deasserting `table[b].cached`.

Another processor may have already induced a flush to processor i, but this introduces no coherence problem, the ejection just expedites the flush. If when the flush request arrives at processor i, it does not find the block in its cache, the processor should not be alarmed. The flush message should simply be discarded.

RO: `(cachetable[i][b mod Ncb].dirty = false)`

When processor i ejects a clean blocks from its private cache, the next pointer of the previous processor must be adjusted so that it no longer points to i, but rather to processor cachetable[i][b mod Ncb].next. Since this list is only singly linked, the only way to update this field is to send a message to the memory module responsible for b, which in turn initiates a traversal of the linked list. Once the walking message reaches the processor immediately before i, call it k, `cachetable[k][b].next` is updated to point to `cachetable[i][b].next` -- effectively removing processor i from the list.

Another processor may have already initiated an invalidation sequence to this list when processor i's ejection occurs, causing a break in the list. To resolve this complication, a counter, called the stream counter and a single bit, called the null received bit, are associated with each of the blocks which was cached RO and is currently being invalidated. These counters and null received bits are held at the memory modules. The number of counters and reserved bits should be sufficient, so they do not become a bottleneck. Also when a processor ejects a clean block, b, it must send, along with ejection request, `cachetable[i][b mod Ncb].next`, so if the processors deeper than i need to be invalidated, they can be reached. When an invalidation sequence is initiated, the null received bit is cleared, and the the stream counter is set to one.

Every time an invalidation sequence terminates, one of two types of messages is sent to the memory module. If the invalidation sequence terminates because an invalidation message was sent to a processor which did not have the block anymore, a plain termination message is sent to the memory module. If the invalidation sequence terminated because a NULL pointer was encountered, a termination by null message is sent the the memory module. When the memory module receives either of these messages it decrements the corresponding stream counter. If the message is a "terminated by null" message, the null received bit is asserted as well. Block ejection messages received by a memory module are cross-referenced against the list of blocks currently undergoing invalidations; if the block is currently being invalidated, an invalidation sequence is initiated with processor `cachetable[j][b mod Ncb].next` (j is the ejecting processor), and the stream counter is incremented.

Only when the stream counter equals zero and the null received bit is clear is the write performed with respect to all processors. At this point, the processor which initiated the write operation should receive a message indicating the write is performed.

Note that in both the RO and RW cases the private caches do not need to wait for a response from main memory granting permission to eject the block -- this is very crucial, for block ejections are too frequent for the system to tolerate the delay involved with waiting for an ejection confirmation from the memory module, or another processor.

6.3. Multiple Singly Linked List Protocol

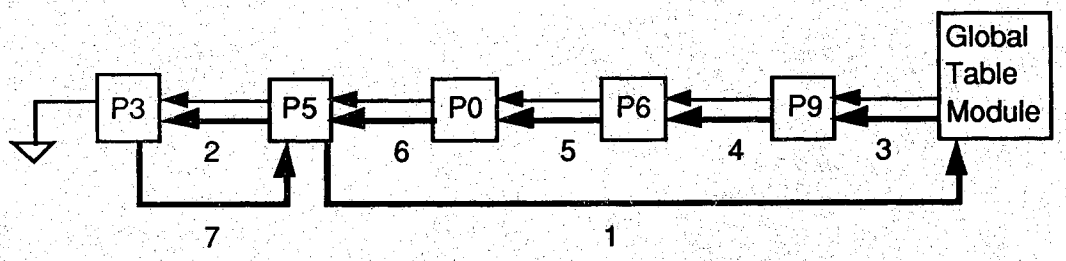
Using a broadcast mask to represent the start of the list is one way to eliminate having to walk the linked list on ejections. When processor i ejects block b, `cachetable[i][b].next` is sent to the memory module, and is merged with the previous head of the list. When it comes time to invalidate the present set, an invalidation sequence is set to each of the destinations implicit in the broadcast mask. This way, ejections don't need to update the linked list, rather just the start of the list. This idea can be extended to include any kind of fixed length entry format.

The problem with this crutch is the performance degradation. If many ejects are done before a write is performed, the method degrades to a fixed length entry, single broadcast mask method, as seen in chapter three.

6.4 Limiting the Number of Shared Blocks

If the number of shared blocks (Nsb) accessed by a processor is limited to the capacity of the private caches (Ncb), and the shared blocks are placed in contiguous locations, there will not be two shared cache blocks contending for the same block in the private caches. This simplifies the protocol because block ejections no longer need to update the linked list. With no block ejections the number of messages for private cache operations is reduced to fewer messages than that required for non-linked list protocols. Furthermore, the stream counters and the null received bits can be eliminated.

Write misses and write hits can also be implemented more efficiently. For the case of write hits, consider the same example just discussed. The writing processor, five, can initiate an invalidation sequence to processor three directly.



Explanation of Messages

1) Processor five sends a message to the memory module to start invalidating the processors from the start of the list.
2) Processor five sends a message to processor three to start invalidating the end of the list.
3-5) Other processor's copies are invalidated.
6) Processor five is notified that the beginning section of the list is invalidated.
7) Processor three notifies processor five that the last section of the list is invalidated.

Figure 6.7 Write Hit by Processor Five to Shared Block (No Ejection)

Write misses can also be handled differently. The last processor in the list can directly send a message to the referencing processor, rather than sending a message to the global table which in turn sends a message to the referencing processor.

The number of messages for this linked list protocol is less than that required for two counter systems. Keep in mind that |P| is really $|P_{sloppy}|$ because there are no block ejections. The P for double counter systems will vary depending upon whether or not tidy ejection is employed, and on how P is represented.

Operation	Number of Messages
Read Miss	2
Write Miss	
Not Cached	2
Cached RO	|P| + 3
Cached RW	4
Write Hit	
Cached RO	|P| + 3

Figure 6.8 Number of Messages in Singly Linked List Protocol With Limited Nsb

Unfortunately, the linked list methods sequentialize the invalidations of the processors, making the time to perform writes longer. The only time this causes performance degradation is when synchronization variables are to be accessed, and the writes issued by the processor wanting to access a synchronization variable are not performed with respect to all processors. Consequently, linked list protocols excel when accesses to synchronization variables are infrequent. They take advantage of the reduced network traffic during the duration of execution which doesn't access synchronization variables. Furthermore, compiler assists might be done to move the last write before an access to a synchronization variable so it is issued as soon as possible.

6.5 Doubly Linked Lists

Because it is undesirable to bound the number of shared blocks to the number of private cache blocks, and require that the shared blocks be placed in contiguous locations, we invented doubly linked list protocols. Changing the linked list organization to a doubly linked list eliminates having to walk the linked list when blocks are ejected -- only the pointers at the adjacent processors in the list need to be updated. In order to implement doubly linked list, the cache tag format must be changed. Figure 6.9 shows the updated global table format.

```
type
   processor_number_type = 1..N
   tag_type = 1..MAXTAG
   entry_type =   record
                     cached:boolean;
                     rw:boolean;
                     head:processor_number_type;
                  end
   cache_entry_type = record
                         dirty:boolean;
                         nullnext,nullprev:boolean;
                         next,previous:processor_number_type;
                         ejecting,grantedejection:boolean;
                         tag:tag_type;
                      end
var
   table:array[1..M] of entry_type;
   cache_table: array[1..N][1..Ncb] of cache_entry_type;
```

Figure 6.9 Software Analogy for Double Linked Lists

In order to maintain the previous pointer, a message must be sent to the processor pointed to by `table[b].head`, whenever a read miss occurs. This message must carry the processor number of the processor which has just had the read miss. The first processor in the list updates its previous pointer when it receives this message.

6.5.1 Simultaneous Ejection Problem

When a block is ejected, the update messages cannot be indiscriminately sent to the adjacent processors. Consider the example where processors five and two both decide to eject block b concurrently. If processor five sends messages to processors six and two, and processor two sends messages to processors five and three, the linked list will become corrupted. Processor three's and processor six's pointers will point to processors which no longer have the block, and nothing will be done with the messages sent to processors two and five. Figure 6.10 shows this simultaneous ejection problem.

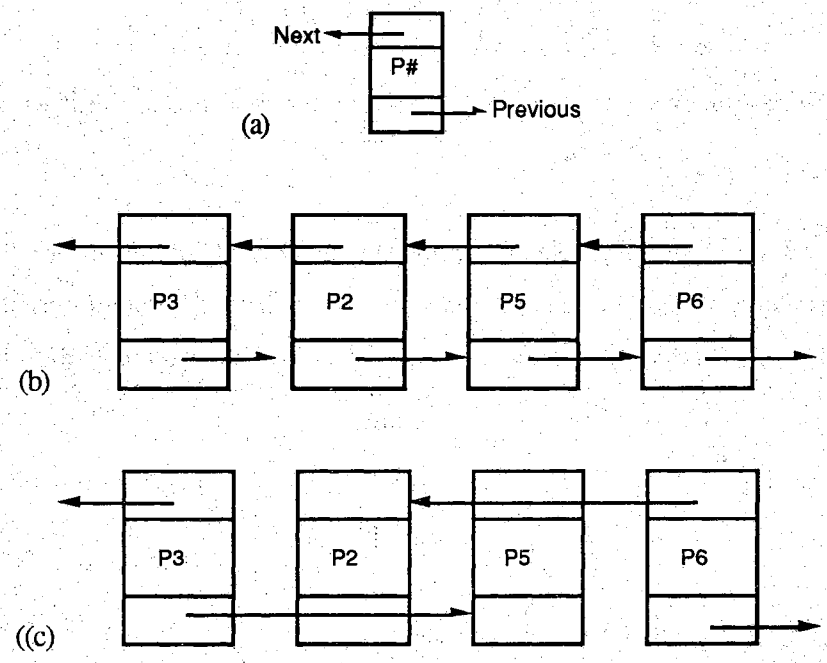


Figure 6.10 Simultaneous Ejection Example (a) A Single Processor (b) Before (c) After

In order to insure the list does not become corrupted, **ejection request** messages can be sent to the adjacent processors to make sure they are not currently ejecting the same block. The algorithm followed by a processor i trying to eject a block b is shown in Figure 6.11.

```
while block not ejected do
begin
   while (cache_table[i][b].grantedejection == true)
   {Already granted permission to the adjacent processor}
   begin
      wait;
   end
   cache_table[i][b].ejecting = true;
   send ejection request messages to next and prev processors;
   if both messages generate affirmative responses
   begin
      eject the block;
      send cache_table[i][b].next to cache_table[i][b].prev;
      send cache_table[i][b].prev to cache_table[i][b].next;
   end
   else
   begin
      if(cache_table[i][b].next granted)
      begin
         send previous pointer update message where
         the new previous value is i;
         {Restore the original previous pointer}
      end
      if(cache_table[i][b].prev granted)
      begin
         send next pointer update message where
         the new next value is i;
         {Restore the original next pointer}
      end
      cache_table[i][b].ejecting = false;
   end
end
```

Figure 6.11 Algorithm for Ejection of Shared Blocks

The algorithm for receiving ejection requests is shown in Figure 6.12, where i is the ejecting processor number, and j is the processor receiving the ejection request.

```
switch(message_type)
begin
   case ejection request:
      if(cache_table[j][b].ejecting == false)
      begin
         cache_table[j][b].grantedejection = true;
         send affirmative response;
      end
      else
         send negative response;
   case update previous:
      cache_table [i][b].prev = newprevious;
      cache_table [i][b].grantedejection = false;
   case update next:
      cache_table [i][b].next = newnext;
      cache_table [i][b].grantedejection = false;
end
```

Figure 6.12 Algorithm for Receiving Ejection Requests

It is not possible for a set of processors to get into deadlock waiting for each other to eject a block. A deadly embrace is avoided because the necessary condition **hold and wait** is never satisfied. That is, a processor never asserts `cache_table[i][b].ejecting` and from then on waits for its adjacent processors to grant permission to eject. Once one of the adjacent processors rejects permission, `cache_table[i][b].ejecting` is cleared, and the processor re-requests after some amount of time. In a bad and unlikely case, it is possible for the ejection to be delayed for a long time because two adjacent processors are simultaneously trying to eject the block and end up colliding several times before one of the processors successfully ejects the block.

6.5.2 Expediting Invalidation for Doubly Linked List Protocols

In doubly linked list systems, write hits may be hurried by invalidating the list in three rather than two directions. To understand the performance increase due to this enhancement, consider the linked list as a line segment [0,|P|], and the processor performing the write as a randomly chosen point, s, on the line. The time is takes to perform the write is proportional to the longest of the line segments [0,s] and [s,|P|] in the singly linked list case. The problem can be made tractable if the probability of the ejecting point is assumed to be uniformly distributed over the line. In our idealized system:

E{number of sequential network transfers to perform a write hit in a singly linked system} =

$$\int_0^{|P|} \max(s,|P|-s)\frac{1}{|P|}\,ds = \int_0^{\frac{|P|}{2}} \frac{|P|-s}{|P|}\,ds + \int_{\frac{|P|}{2}}^{|P|} \frac{s}{|P|}\,ds = \frac{3}{4}|P| \tag{6.1}$$

When the linked list is a double linked list the invalidation can be done in three direction, rather that two. The addition direction is from the writing processor to the memory module. Again turning to the idealized system, the time to perform a write hit can be estimated.

E{number of sequential network transfers to perform a write hit in a double linked system} =

$$\int_0^{|P|} \max(s,\frac{|P|-s}{2})\frac{1}{|P|}\,ds = \int_0^{\frac{|P|}{3}} \frac{|P|-s}{2|P|}\,ds + \int_{\frac{|P|}{3}}^{|P|} \frac{s}{|P|}\,ds = \frac{16}{36}|P| \tag{6.2}$$

This requires more messages, and there is more return receipt contention at the writing processor's interconnection network port. In fact, non-linked list systems are the extreme case of this invalidation parallelism.

6.6 Using Backup Tag Entries

A way to make ejections less frequent is to add one or more extra fields to the cache tags allowing them to store multiple pointers for each cache block. This way, the only time ejected blocks must use the interconnection network is when all the backup locations are being used to store previously ejected global blocks, and the block which is being brought into the cache is also global. Figure 6.13 shows how the tag memory can be modified

```
cache_entry_type = record
                       next_backup:processor_number_type;
                       prev_backup:processor_number_type;
                       tag_back:tagtype;
                       nullnext,nullprev:boolean;
                       next,previous:processor_number_type;
                       ejecting,grantedejection:boolean;
                       tag:tagtype;
                   end
```

Figure 6.13 Adding Backup Locations

When the backup locations are not being used, `next_backup != prev_backup`. This alleviates the need for a back up location "in use" bit.

The number of backup locations needed to insure that no block ejections occur is large if the shared blocks are not constrained to contiguous locations. The worst situation is when all shared blocks map to the same private cache block, thus requiring Nsb backup locations. This is an unlikely event. Using analytical analysis, the probability that bringing in another shared block will require the freeing up of a backup location can be estimated, for a given number of shared blocks, cache blocks, and backup locations. In the worst case, all of the shared blocks are <u>not</u> being written to, so all processors cache all shared blocks. In this case, each shared block requires a backup location. Assuming that every processor is equally likely to cache a block, and each cache set is equally likely to receive the block, the probability that a miss to a shared block requires freeing up backup locations can be analytically estimated. Consider a representative cache block, and imagine the mapping of a shared block to a cache set as a trial. If shared blocks are equally likely to map into any cache set, the probability that a shared block will map to the representative set is 1/Ncb. The probability that exactly k of the Nsb shared blocks will map to the representative cache block is governed by a binomial distribution.

P{A private cache block will require exactly k backup locations to hold all shared blocks which map to it } =

$$\mathrm{Bin}(k,\frac{1}{Ncb},Nsb) = \frac{Nsb!}{k!(Nsb-k)!}\left[\frac{1}{Ncb}\right]^k\left[1-\frac{1}{Ncb}\right]^{Nsb-k} \qquad (6.3)$$

The probability that a block will need k or fewer backup locations to store all the pointers for its shared blocks is:

P{A private cache block requires k or fewer backup locations to hold all the shared blocks which map to it} =

$$= \sum_{j=k}^{Nsb} \frac{Nsb!}{j!(Nsb-j)!}\left[\frac{1}{Ncb}\right]^j\left[1-\frac{1}{Ncb}\right]^{Nsb-j} \qquad (6.4)$$

If there are many (>30) shared blocks and cache blocks, this distribution can be approximated as a normal distribution with mean Nsb/Ncb and variance Nsb(1-1/Ncb)/Ncb.

In order to make sure the backup locations don't all fill up, causing all ejections to operate at the degenerate performance, the backup locations need to freed at the end of an application. This causes a burst of network traffic whenever a process terminates. If the system is not multitasking, the caches could be designed so that all the backup locations in a single private cache could be invalidated at once.

CONCLUSIONS

Throughout the thesis, ideas have been presented on how to make directory based protocols use less memory and generate fewer interconnection network messages.

To reduce the size of the global table without inducing an unacceptable amount of network traffic, alternate entry formats were invented and analyzed. The compromises between the Archibald and Baer entry format and Censier and Feautrier entry format looked at were: a single broadcast mask, multiple broadcast masks, and grouped bit vector representations. In order to determine the expected number of redundant or extraneous messages, a Markov chain for RW cache blocks was developed. This chain has its transitional probabilities expressed in terms of program and system attributes, making it easier to see what parameters are important for cache effectiveness. The steady state probabilities of the RW chain were found to be most effected by the fraction of writes. After modeling the behavior of shared block and these alternate formats, it was discovered that the difficulties in coming up with an optimal covering of the present set cause multiple broadcast mask systems to be less accurate than grouped systems with a comparable number of bits per entry.

An economical way of implementing queuing semaphores was developed Using a software simulation, the two priority scheme was found to be just as fair as first in first out. No additional waiting time is introduced by the two counter method.

In addition to the Markov chain for RW blocks, Markov chains were developed for RO and sloppily ejected blocks. A closed form solution for RO block was also developed; no one parameter dominates the steady state probabilities of this chain. From the sloppy ejection chain, it was seen that not updating the global table on ejections of clean blocks is desirable.

Rather than changing the entry format for the global table, because this always leads to some extraneous and redundant invalidations, the table can be made to record elaborate entries for

only those shared blocks which are currently cached. A pipelined architecture of a variable length table was described and evaluated using a scheduling simulator to determine the speedup brought about by pipelining. The simulator randomly generates global table operations (like write hits) as a Poisson process with a specified rate, and schedules them according to a greedy strategy. Pipelining the global table yielded a maximum speedup of 1.7.

The last section presented several variations of linked list protocols which reduce the amount of interconnection traffic. A singly linked protocol which supports block ejection was defined, but this protocol is not effective because of the network communication created by block ejections. Constraining the shared data to a fixed number of contiguous blocks was found to reduce the network traffic to less than that required by two counter systems. Finally, doubly linked list protocols were defined. The advantage of doubly linking the list is that ejecting a block requires only cooperation with the two adjacent processors, making it possible to have some of the traffic reduction of the singly linked protocol without the limited number and contiguously placement restrictions on shared blocks.

BIBLIOGRAPHY

[AgCh89] Agarwal, A. and Cherian M., "Adaptive Backoff Synchronization Techniques," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 396-406, 1989.

[ArBa85] Archibald, J. and Baer, J.L., "An Economical Solution to the Cache Coherence Problem," *Proceedings of the 12th Annual International Symposium on Computer Architecture*, pp. 355-362, June 1985.

[ArBa86] Archibald, J. Baer, J. L., "Cache Coherency Protocols: Evaluation Using a Multiprocessor Simulation Model," *ACM Transactions on Computer Systems*, pp. 273-298., November 1986.

[ASHH88] Agarawal, A., Simoni, R., Hennessy J., and Horowitz M., "An Evaluation of Directory Schemes for Cache Coherence," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 280-289, 1988.

[Baas78] Baase, S., *Computer Algorithms Introduction to Design and Analysis*, Addison Wesley, 1978.

[BaRa89] Baylor S. and Rathi B., "A Timestamp Cache Coherence Scheme," *Proceedings of the International Conference on Parallel Processing*, pp. 24-32, Vol. I, 1989.

[BaWa88] Baer, J.L. and Wang, W.H., "On the Inclusion Property for Multilevel Cache Hierarchies," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 73-80, 1988.

[BBN85] BBN Butterfly Parallel Processor Overview, 1985.

[BHMS85] Brayton, R.K., Hackel, G.D., McMullen C.T. and Sangiovanni-Vincentelli, *Logic Minimization Algorithms for VLSI Synthesis*, Klower Academic Publishing, 1985.

[BrDa77] Briggs, F. and Davidson,E., "Organization of Semiconductor Memories for Parallel Pipelined Processors," *IEEE Trans. of Computers*, C26, Feb. 1977.

[BrHo90] Brooks, E. and Hoag, J., "A Scaleable Coherent Cache System with Fuzzy Directory State," Submitted to *Proceedings of the International Conference on Parallel Processing*, August 1990.

[CeFe78] Censier, M., and Feautrier, P. "A new Solution to Coherence Problems in Multicache Systems," *IEEE Trans. on Computers*, C27(12);pp. 1112-1118, December 1978.

[ChVe88] Cheong H. and Veidenbaum, A., "A cache coherence scheme with selective invalidation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 299-307, June 1988.

[Davi71] Davidson, E.S., "The Design and Control of Pipelined Function Generators," *Proceedings Int. IEEE Conference on Systems, Networks, and Computers*, Oaxtepec, Mecico, January 1971, pp.19-21.

[DeTh90] Delagi, B. Thaper, Submitted to *IEEE Computer*, June 1990.

[Dubi87] Dubios, M., "Effects of Invalidation on the Hit Ration of Cache-Based Multiprocessors," *Proceedings of the International Conference on Parallel Processing*, pp. 255-257, 1987.

[DuBr82] Dubios, M. and Briggs, F., "Effects of Cache Coherency in Multiprocessors," *IEEE Trans on Computers*, vol. 37, pp. 58-70, November 1982.

[DuSB86] Dubois, M., Scheurich, C., and Briggs, F., "Memory Access Buffering in Multiprocessors," *Proceedings of the 13th Annual International Symposium on Computer Architecture*, pp. 434-442, 1986.

[DuSB88] Dubios, M. Scheurich, C. and Briggs, F., "Synchronization, Coherence, and Event Ordering in Multiprocessors," *IEEE Computer*, pp. 9-21, February 1988.

[EgKa88] Eggers, S. and Katz, R., "A Characterization of Sharing in Parallel Programs and its application to Coherency Protocol Evaluation," *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pp. 373-381, May 1988.

[EgKa89] Eggers, S. and Katz, R., "The Effects of Sharing on the Cache and Bus Performance of Parallel Programs," *Proceedings of APLOPS III*, pp. 257-270, April 1989.

[GGKM83] Gottlieb, A., Grishman, R., Kruskal, C. P., McAuliffe, K.P., Rudolph, L., and Snir, M., "The NYU Ultracomputer -- Designing an MIMD, Shared Memory Parallel Machine," *IEEE Transactions on Computers*, Februrary 1983, pp.175-189.

[GoLR83] Gottlieb, A., Lubachevsky, R., and Rudolph, L., "Basic Techniques for the Efficient Coordination of Very Large Numbers of Cooperative Sequential Processors," *ACM Transactions on Programmign Languages and Systems*, April 1983, pp.164-189.

[Good83] Goodman, J., "Using Cache Memory to Reduce Processor-Memory Traffic," *Proceedings of the 10th Annual International Symposium on Computer Architecture*, pp. 124-131, 1983.

[Hill88] Hill, M., "A Case for Direct-Mapped Caches," *IEEE Computer*, pp.25-39. December 1988.

[Kogg81] Koggee, P., *The Architecture of Pipelined Computers*, McGraw Hill, 1981.

[MBLZ89] Mizrahi, H., Baer, J.L., Lazowska, E., and Zahorjan, J., "Extending the Memory Hierarchy into Multiprocessor Interconnection Networks: A Performance

Analysis," *Proceedings of the International Conference on Parallel Processing*, pp. 41-50, Vol. I, 1989.

[MiBa89] Min, S.L. and Baer, J.L., "A Timestamp Cache Coherence Scheme," *Proceedings of the International Conference on Parallel Processing*, pp. 24-32, Vol. I, 1989.

[Moto88] Motorola 88200 Data Book, Motorola Inc., Austin, Texas, 1988.

[PeSi85] Peterson, J. and Silberschatz, A., *Operating System Concepts*, Addison Wesley, 1985.

[RCCT90] Rettberg, R., Crowther, W., Carvey, P., and Tomlinson, R., "The Monarch Parallel Processor Hardware Design," *IEEE Computer*, pp.18-29, April 1990.

[Sequ87] Sequent Computer System, Inc., Symmetry Technical Summary, Beaverton, OR, 1987.

[Sieg85] Siegel, H.J., *Interconnection Networks for Large Scale Parallel Processing*, Lexington Books, 1985.

[Stre89] Strenstrom, P., "A Cache Consistency Protocol for Multiprocessors with Multistage Networks," *Proceedings of the 16th Annual International Symposium on Computer Architecture*, pp. 407-415, June 1989.

[Tang76] Tang, C.K. "Cache Design in Tightly Coupled Multiprocessor Systems," *Proceeding of AFIPS* , National Computer Conference, June 1976, pp. 749-753.

[WaCa89] Warner, C. and Casavant, T., "Channel Routing on the NCUBE," Independent Study Project Report, Purdue University 1989.

[WaMe90] Wailes, T. and Meyer, D., "Multiple Channel Architecture," Submitted to *Frontiers of Parallel Computing*, 1990.

[WeGu89] Weber, W. and Gupta, A., "Analysis of Cache Invalidation Patterns in Multiprocessors," *Proceedings of APLOPS III*, pp. 243-256, April 1989.

[YaBL89] Yang, Q., Bhuyan, L., and Liu, B.C., "Analysis of Comparison of Cache Protocols for a Packet-Switched Muliprocessor," *IEEE Trans on Computers*, vol. 38 No. 8, pp. 1143-1153, August 1989.