

**LIBRARY
Michigan State
University**

**PLACE IN RETURN BOX to remove this checkout from your record.
TO AVOID FINES return on or before date due.**

DATE DUE	DATE DUE	DATE DUE
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____
_____	_____	_____

MSU Is An Affirmative Action/Equal Opportunity Institution

c:\circ\dtedue.pm3-p.1

DESIGN AND EVALUATION OF A
HIERARCHICAL BUS MULTIPROCESSOR

By
Carl Burton Erickson

A DISSERTATION

Submitted to
Michigan State University
in partial fulfillment of the requirements
for the degree of

DOCTOR OF PHILOSOPHY

Department of Electrical Engineering

1991

Abstract

Design and Evaluation of a Hierarchical Bus Multiprocessor

By

Carl Burton Erickson

This thesis describes the design and evaluation of the protocol and controllers which maintain memory coherence in a hierarchical bus multiprocessor. The architecture studied represents a natural step in the evolution of bus-based shared memory multiprocessors. Extending the scalability of these machines beyond the current single bus limit of 20 processors preserves the inherent advantages and familiarity of the bus interconnect and the shared memory programming paradigm.

Cache memory is vitally important in bus-based multiprocessors for reducing memory latency and for conserving bus bandwidth. A new cache coherence protocol, designed for hierarchical buses, is used to solve the attendant problem of cache and memory coherence. A design and debugging tool based on Petri net simulation was developed to evaluate detailed models of the controllers implementing the cache coherence protocol. The Petri net simulator is completely general, but is particularly adept at representing concurrency and synchronization; a necessity for modeling parallel computer architectures.

Performance evaluation of the architecture under study was accomplished with object-oriented discrete event simulation. The overall performance of a bus-based multiprocessor is heavily dependent on the performance of cache memory, since the system buses are potential bottlenecks. Two simulation models were developed. The first, a purely probabilistic model, can be used to quickly explore a wide range of system configurations and parameters. A second, more detailed, trace-driven model accurately represents the activities and state of a single cluster of processors. Trace-driven simulation is necessary in evaluating the hierarchical bus multiprocessor since

the performance of cache memory is highly dependent on the pattern of memory accesses. A method of gathering architecture independent multiprocessor address traces on a conventional uniprocessor was developed to facilitate trace-driven simulation.

Simulation indicates the hierarchical bus architecture increases the ultimate size of bus-based multiprocessors by nearly an order of magnitude to approximately 200 processors. Detailed Petri net modeling and simulation suggests the feasibility of the controllers and the correctness of the cache coherence protocol.

To my parents,
and Mary Caroline O'Neill

Acknowledgments

This research and the degree it represents would not have been possible were it not for my dual thesis advisors, Dr. Mani Azimi and Professor Lionel Ni. The time spent with Mani in front of workstations, at the chalk board, and over numerous dinners has been distilled into this thesis. For all the knowledge I have absorbed, both technical and non-technical, and for our friendship, I am most grateful. Dr. Ni has been most generous with his time in the final year and a half of my degree; I have benefited in many ways from his considerable experience and knowledge.

I also appreciate the assistance and interest of my other committee members, Dr. Richard Enbody, Dr. Erik Goodman, Dr. John Kreer, Dr. Habib Salehi and Dr. Anthony Wojcik. As chairman of the Electrical Engineering department when I started at Michigan State, Dr. Kreer helped me in many ways, the last being to serve on my thesis committee. Erik gave generously of his time in reviewing my research, and provided financial support through my long and pleasant association with the Case Center.

Working with the numerous people who have been a part of the Case Center for the last five years has been an enjoyable and educational experience. In particular Jackie Carlson, Leslie Hoppensteadt, and Homayoun Torab have made Michigan State a happier place for me.

This research has been supported by the National Science Foundation under grant MIPS 8811815 and by the College of Engineering.

Finally, as the most important people in my life, I acknowledge with love the support of my extended family and in particular my wife, Mary Caroline.

Table of Contents

List of Tables	x
List of Figures	xii
1 Introduction	1
1.1 Need for Parallel Processing	1
1.2 Elements of Parallel Computers	2
1.2.1 Processors	3
1.2.2 Interconnection network	3
1.2.3 Memory	4
1.3 Cache Coherence	5
1.4 Single Bus Multiprocessor	5
1.5 Hierarchical Bus Multiprocessor	6
1.6 Performance Evaluation	8
1.6.1 Queuing networks	8
1.6.2 Discrete event simulation	10
1.7 Thesis Outline	11
1.7.1 Guide to chapters	12
2 Cache Coherence	16
2.1 Definition of Cache Coherence Problem	16
2.2 Example of Cache Coherence Problem	18
2.3 Solutions to the Cache Coherence Problem	20
2.3.1 Software solutions	20
2.3.2 Hardware solutions	21
2.3.3 Single bus snoopy protocols	22

3	Architecture of the HBSM Multiprocessor	25
3.1	Introduction	25
3.2	System Controllers	28
3.2.1	Cache Controller (CC)	30
3.2.2	Memory Controller (MC)	30
3.2.3	Cluster Cache Controller (CCC)	31
3.2.4	Cluster Memory Controller (CMC)	33
3.2.5	CCC/CMC state mapping	35
3.3	Interference Events	36
3.3.1	Collision events	37
3.3.2	Benign events	38
3.3.3	Coherence events	38
3.3.4	CMC and interference events	39
3.3.5	CCC and interference events	39
3.4	Bus Arbitration	41
4	COGI Cache Coherence Protocol	42
4.1	Design of COGI Protocol	42
4.2	COGI Protocol Tables	43
4.3	Example of COGI Protocol	53
4.4	Other Protocols	57
5	Multiprocessor Address Traces	58
5.1	Introduction	58
5.1.1	Tracing techniques	60
5.2	The Traced Application	61
5.3	Trace Generation	64
5.3.1	Expanding assembly language	64
5.3.2	Compile and execute	67
5.3.3	Filter raw traces	67
5.4	Extension of Tracer	70
6	Petri Net Model	73
6.1	Petri Net Modeling	73
6.1.1	Extensions	75

6.1.2	Software packages	76
6.2	High Level Timed Petri Nets	77
6.2.1	Places	77
6.2.2	Directory places	77
6.2.3	Transitions	78
6.2.4	Connectivity	79
6.3	HLTPN Simulator	79
6.4	HBSM Multiprocessor Model	81
6.4.1	Processors	81
6.4.2	Synchronization	83
6.4.3	Bus arbitration	85
6.4.4	Directories	87
6.5	Design and Debugging	90
6.5.1	Correctness	93
7	Discrete Event Simulation	94
7.1	Introduction	94
7.2	Simulation Environment	97
7.2.1	Multiprocessor models	99
7.2.2	System parameters	101
7.2.3	Measures of performance	102
7.2.4	Correctness	104
7.3	Probabilistic Model	107
7.3.1	Single cluster	109
7.3.2	Multiple cluster	115
7.4	Trace-driven Model	121
7.4.1	Intercluster activity	123
7.4.2	Benchmark parallel merge sort	126
7.4.3	Results for 8×8 system	127
7.4.4	Cache geometry	132
7.4.5	Coherence traffic	134
7.5	Summary	140

8	Conclusions and Future Research	143
8.1	HBSM Multiprocessor Architecture	143
8.1.1	Performance improvements	145
8.1.2	System requirements	146
8.2	High Level Timed Petri Net Simulator	147
8.3	Simulation Methodology	148
	Bibliography	150

List of Tables

2.1	Important stages in the evolution of snoopy cache coherence protocols.	23
3.1	The state of blocks in the CC.	31
3.2	The state of blocks in the CCC.	32
3.3	The state of blocks in the CMC.	34
3.4	Mapping of CMC states to CCC and CC.	36
3.5	Mapping of CCC states to CMC and CC.	36
3.6	Coherence events detected by the CCC and their resolution.	41
4.1	Transaction types on the cluster bus and processor events.	44
4.2	Transaction types on the global bus.	44
4.3	Actions of CC in response to processor requests.	46
4.4	Actions of CC in response to cluster bus transactions.	47
4.5	Actions of CMC in response to cluster bus transactions.	48
4.6	Actions of CMC in response to global bus transactions.	49
4.7	Actions of CCC in response to cluster bus transactions.	50
4.8	Actions of CCC in response to global bus transactions.	51
4.9	Actions of MC in response to bus transactions.	52
5.1	Tracer library functions for barrier synchronization and tracing. . . .	63
7.1	Technology dependent system parameters for simulation model. . . .	101
7.2	Probabilistic representation of memory accesses.	102
7.3	Techniques of increasing confidence in correctness of models.	106
7.4	Probabilistic parameters for simulation of first model.	108
7.5	Effect of hit ratio on multiprocessor cycles per instruction (MCPI) for single cluster multiprocessor Experiment 1.	112

7.6	Multiprocessor cycles per instruction (MCPI) for ideal application Experiment 5.	118
7.7	Multiprocessor cycles per instruction (MCPI) for non-ideal application Experiment 6.	120
7.8	Cases of intercluster transactions in COGI protocol.	125
7.9	Experiment 7 multiprocessor cycles per instruction (MCPI) data for set size 2.	133
7.10	Experiment 7 distribution of cluster bus busy time.	139

List of Figures

1.1	Generalized two-level hierarchical bus multiprocessor.	7
1.2	Steps in the evolution of bus-based shared memory multiprocessors. .	13
2.1	Simple multiprocessor scenario to demonstrate cache coherence problem.	19
3.1	Components of the Hierarchical Bus Shared Memory (HBSM) multi-processor architecture.	27
3.2	Detail of the CCC dual-bus snoop device showing the two active portions of the controller, cluster snoop and global snoop, sharing access to the single cache directory.	29
3.3	Meaning of the block states of the CCC.	32
3.4	Meaning of the block states of the CMC.	34
4.1	Basic scenario of intracluster sharing of locally resident block. Processor P_1 reads block, P_{10} reads then writes block, passing updated version of data word to P_1	54
4.2	Continuation of the COGI protocol example transaction. Processor $P_{2,1}$ reads then writes the same block that is shared with Cluster 1. .	56
5.1	Traced multiprocessor address space. Raw virtual references are shown with resulting filtered references below them and indented.	62
5.2	C template for process creation and synchronization.	65
5.3	The steps of the Tracer technique.	66
5.4	Expanded assembly code (in two columns) shown below original assembly code fragment.	68
5.5	Problem of extending Tracer to general synchronization and task allocation paradigm.	71
6.1	Main loop of HLTPN simulator.	82

6.2	HLTPN model representing the actions of processors in the hierarchical bus multiprocessor.	84
6.3	HLTPN subnet of barrier synchronization in the hierarchical bus multiprocessor model.	86
6.4	HLTPN subnet of bus arbitration for HBSM multiprocessor.	88
6.5	Set associative and direct mapped directories for a 2 cluster, 2 processor/cluster hierarchical cache system model.	89
6.6	Bus handshaking signals DI and DK in Petri net model.	92
7.1	Simulation loop for event scheduled DES using C-like pseudo code. . .	96
7.2	Effect of hit ratio on cluster bus utilization for single cluster multiprocessor.	109
7.3	Effect of global accesses on cluster bus utilization for single cluster multiprocessor.	111
7.4	Effect of dirty write-back probability on cluster bus utilization in single cluster multiprocessor.	113
7.5	Effect of write notice probability on cluster bus utilization in single cluster multiprocessor.	114
7.6	Global bus utilization in ideal application.	116
7.7	Potential efficiency for ideal application.	117
7.8	Global bus utilization for realistic application.	119
7.9	Potential efficiency for realistic application.	121
7.10	The benchmark parallel merge sort used for traces in the hybrid model.	128
7.11	Bus utilizations in typical configuration of block size 4, set size 2. . .	129
7.12	Hit ratio and potential efficiency in typical configuration of block size 4, set size 2.	130
7.13	Effect of block size on potential efficiency for set size 2.	133
7.14	Effect of set size on potential efficiency for optimal block size. . . .	134
7.15	Effect of set size on global bus utilization for block size 4.	135
7.16	Fraction of cache writes which generate a cluster bus write notice. . .	136
7.17	Fraction of write notices to blocks which are not shared by other caches.	137
7.18	Fraction of read misses which are serviced by another cache with a dirty copy of the block.	138

Chapter 1

Introduction

This thesis describes the design and performance of the hardware controllers and cache coherence protocol which together comprise the memory subsystem of a hierarchical bus shared memory multiprocessor. The architecture studied represents a logical evolutionary step in the inevitable ascendancy of parallel processing in computationally intensive application domains such as scientific computing. The inevitable need for parallel computers and the basic elements of these machines is outlined in this introductory chapter. The first generation of single bus multiprocessors are considered, and the natural extension of a hierarchical bus architecture is described. The contributions of this thesis are a solution to the problem of cache coherence in a hierarchical bus multiprocessor and performance evaluation techniques for trace-driven simulation. The final section summarizes the strategy underlying this work and the contents of the remaining chapters.

1.1 Need for Parallel Processing

The eventual use of parallel processing technology is assured by the limitless need for computational power and the inherent physical limits of sequential computers. The areas of research that could benefit from faster, more efficient computers is as large as the number of fields being studied with computers. Classic examples of applications which are limited by available computing power include numerical weather forecasting, computational fluid dynamics, finite element analysis, and image processing. The future will undoubtedly bring uses for the computer which are as yet unimagined, and which will require vast computational power.

The computational power of traditional von Neumann architecture uniprocessor computers is limited by the speed of light. Signal transmission speed in silicon is actually ten times slower than the speed of light in a vacuum ($3 \times 10^8 m/sec$) [18]. At this rate, a signal can only propagate 3 cm in $10^{-9} sec$. As the clock rate of the fastest supercomputers [15] approaches $10^{-9} sec$, practical constraints on packaging and cooling

will begin to limit their computational power. If such supercomputers were restricted to purely sequential architectures, then executing one floating-point operation per clock cycle would imply a limit of 1000 MFLOPS (millions of floating-point operations per second). In fact, extant supercomputers, and those announced for the near future, rely on low-level parallel processing techniques known as pipelining and vector processing to increase their maximum potential speed beyond 1000 MFLOPS. Materials more exotic than silicon (such as gallium arsenide) may have a smaller propagation time, but are nevertheless eventually limited by the speed of light.

Where will computer engineers of the future find the performance improvements necessary to satisfy the voracious appetite for computational power of scientific applications? Whatever the technology, if the speed at which information can be processed is limited, then the only method of processing a constant amount of information in less time is to perform multiple operations simultaneously. Coordinated, simultaneous work by multiple processors is the essence of parallel processing. Implementations of parallel computers range from large numbers of slow processors to small numbers of fast processors. Hennessy and Patterson [30] refer to the El Dorado¹ of parallel processing as large numbers of powerful processors in the same computer. The two distinct roads to this El Dorado are exemplified by the 8 processor Cray Y-MP from Cray Research and the 65,536 bit-serial processor CM-2 from Thinking Machines.

The key to parallel architectures which follow the Cray Y-MP path is *scalability*. In practical terms, scalability is usually judged over a range. For example, binary hypercube multicomputers may scale well over the range of 10 to 10,000 processors, but packaging, cooling, and other mundane constraints may make hypercubes impractical outside of this range. Parallel computers which rely on a shared bus as an interconnection network have shown very limited scalability. Such computers are limited by bus bandwidth to around 30 processors; in other words, single bus parallel computers are scalable over less than an order of magnitude.

1.2 Elements of Parallel Computers

The fundamental elements of parallel computers are processors, memory, and an interconnection network. The interconnection network serves to connect processors

¹El Dorado refers to a city of fabulous riches in South America for which Spanish explorers of the 16th century searched in vain.

to other processors as well as to memory modules.

1.2.1 Processors

Processing elements in parallel computers range from the most powerful pipelined vector processors (Cray Y-MP, IBM 3090) made with expensive, fast bipolar semiconductor technology, to the bit-serial, slow, CMOS processing elements of the CM-2. In between these extremes are the high performance RISC microprocessors, such as the Intel i860, Motorola 88000, Sun Microsystems SPARC, and MIPS R3000. Supercomputer processors are designed to be as fast as possible with almost no regard for expense. They are heavily pipelined and support vector instructions. Clock cycle times in such machines are now around $4 \times 10^{-9} \text{sec}$, with sustained performance around 20–30 MFLOPS. Such processors often require elaborate packaging and cooling support, and are never made in large quantities. Most of the supercomputers offered today, and all that are being planned, are available in models with multiple processors [15].

The single chip, VLSI microprocessors with around 10^6 transistors offer the most economical method of building powerful parallel computers with hundreds to thousands of processors. They often include one or more of on-chip data and instruction caches, floating point and integer processors, and memory management units. Mass production of these off-the-shelf components reduces their price, and they generally do not require any special cooling or packaging. Their clock cycle times range from 12.5×10^{-9} to $40 \times 10^{-9} \text{sec}$, and their sustained performance ranges between 2–8 MFLOPS.

1.2.2 Interconnection network

Of the three elements of any parallel computer, the interconnection network is likely the hardest to make scalable. The methods proposed for connecting processors and memories are numerous. Some common interconnection networks include bus, ring, star, tree, mesh, torus, crossbar, and hypercube. Each of these (and the many other possibilities) can be classified by their connectivity, degree, and cost. The bus and crossbar are fully connected with respect to memory; each processor is directly connected to every memory module. The bus has lower cost and lower performance than the crossbar, since each processor node has only one connection (degree one)

and all processors share the same path to all memory modules. With a crossbar, on the other hand, each processor has a direct connection to each memory module, thereby increasing the degree of each node, the cost, and hopefully the performance. In general, increasing the connectivity or the degree of the interconnection network increases the potential performance of the system at the expense of greater complexity and cost.

1.2.3 Memory

Parallel processors can be described by two views of memory: hardware and software. Physically, memory in a parallel processor can be distributed or centralized. Centralized memory may become the bottleneck of the entire computer, limiting the scalability of the architecture. From a software perspective, memory can either be shared or non-shared. Shared memory is seen as the more desirable programming paradigm, since interprocessor communication can take place within the single, shared address space. Non-shared memory programming involves explicit forms of communication (message passing) and can be efficiently emulated on a shared memory machine. Gordon Bell coined the terms *multiprocessor* to refer to shared memory parallel processors and *multicomputer* to refer to non-shared memory parallel processors [7]. Bell termed bus-based multiprocessors built from powerful microprocessors *multis* and predicted that these machines would represent the fifth generation of computing.

The disparate progression of memory and processor technology has resulted in a memory latency problem [30]. The speed of dynamic random access memory (RAM) has grown steadily at a 7% annual rate since 1980. Over the same time period, processor speed has increased at a rate of 19–26% before 1985 and 50–100% rate after 1985. The solution to this mismatch in speed has been a memory hierarchy. Small, fast static random access memories known as caches are placed close to the processor, and serve as a buffer to the larger, slower dynamic RAMs used in main memory. Computer programs generally exhibit both spatial and temporal locality, that is, the memory locations that will be accessed in the near future have a high probability of being physically close in the address space (or the identical) to those accessed in the recent past. The principles of locality assure that a large proportion of memory accesses will be satisfied by the faster but smaller cache, thus decreasing the overall memory latency.

1.3 Cache Coherence

Computers with cache memories may have multiple copies of the same block of memory existing at the same time. These multiple copies of a memory block lead to the problem of cache coherence. A scheme for preserving the correctness of the system in the face of multiple copies of memory blocks is known as a cache coherence protocol. On a uniprocessor, a cache coherence protocol may be as simple as requiring that blocks in the cache which have been modified be written back to main memory before they are replaced in the cache. Input/output operations may also affect cache coherence in a uniprocessor. Cache coherence in a multiprocessor is more complicated, because copies of blocks may exist in multiple caches as well as main memory. The actions of all processors must be coordinated to maintain memory and cache coherence.

Centralized solutions to the multiprocessor cache coherence problem (*e.g.* a single table and controller) are undesirable because the centralized resource is likely to become a system bottleneck. A distributed solution to the cache coherence problem requires that each processor know what actions every other processor is taking, and respond accordingly. A major advantage of the bus-based multiprocessor class identified by Bell is that it admits a distributed, hardware solution to the cache coherence problem. The set of snoopy cache coherence protocols hinges on the ability of each processor to react to the transactions of every other processor by monitoring the bus. By snooping on the bus, each processor receives the same information at the same time, and can take actions to preserve coherence based on this information and the cache coherence protocol. The bus serves to serialize access to the memory assuring that no more than one transaction can take place at any given time, and that all processors have the same knowledge of the system.

1.4 Single Bus Multiprocessor

The single shared bus was a natural choice of interconnection network for the first multiprocessors. The shared bus was inexpensive, fully connected, and a known technology. Computer engineers were familiar with existing bus standards and interfacing techniques. Multiprocessors built on shared buses could be sold in small configurations and later upgraded by adding processors. The full connectivity of the

bus allowed for simpler, centralized memories. Centralized memories, in turn, made the shared memory software paradigm easier to implement. Early commercial multiprocessors built on a single bus include the Symmetry and Balance from Sequent Computer Systems, the Multimax from Encore Computer, the N+1 from Synapse Computer, FX/80 from Alliant Computer Systems, the Sequoia from Sequoia Systems, and the System 64000 from ELXSI. More recently, shared memory bus-based multiprocessors have been used in powerful graphics workstations from Stardent and Silicon Graphics, and as compute and file servers by MIPS, Solbourne, Compaq Computer, and Digital Equipment Corporation.

In addition to reducing memory latency, cache memory plays an equally important role in bus-based multiprocessors by reducing accesses to memory. Single bus multiprocessors without caches would be limited in size to only a few processors. If every memory access by every processor had to go across the single shared bus, the bus would soon become the system bottleneck. Instead, by buffering the bus from every memory access, caches allow the available bus bandwidth to be shared among more processors. Even with cache, the ultimate size of single bus multiprocessors is between 20 and 30 processors. Scalability after this point is limited by the bandwidth of the bus.

1.5 Hierarchical Bus Multiprocessor

The use of caches to reduce traffic on the bus increased the ultimate size of single bus multiprocessors an order of magnitude, from 2–3 processors to 20–30 processors. A logical means of further increasing the scalability of bus-based multiprocessors is to form a hierarchy of single bus multis. A two level hierarchical bus multiprocessor would have several clusters of processors joined by a single global bus. Each cluster would be equivalent to a single bus multiprocessor. Memory could be either distributed among clusters or centralized on the global bus. Figure 1.1 shows the general arrangement of processors and memory in a two-level hierarchical bus shared memory (HBSM) multiprocessor. Processors are gathered into clusters organized around a single bus; memory is distributed throughout the entire system. Such a machine should be able to scale into the hundreds of processors, thus increasing the ultimate size of bus-based multiprocessors an additional order of magnitude.

The Cm* multiprocessor developed at Carnegie Mellon in the 1970s is the most

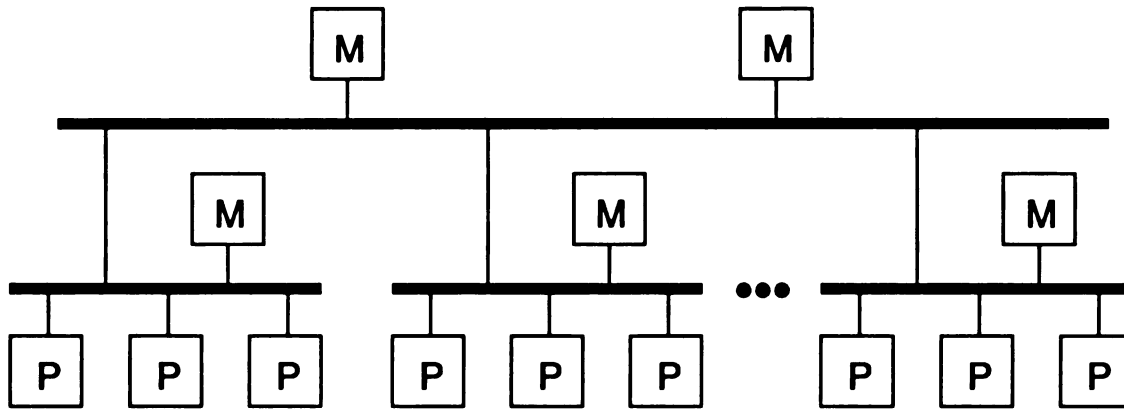


Figure 1.1: Generalized two-level hierarchical bus multiprocessor.

historically significant hierarchical bus architecture. The Cm* was made up of clusters of up to 14 processor/memory modules sharing a single bus and a custom controller. Clusters were joined to one another with two global buses. The custom controller maintained the illusion of a single shared memory address space. Any of the DEC LSI-11 processors could access any other processor's memory module, but the memory latency ratio of 1:3:9 (local:same cluster:different cluster) placed a premium on intercluster accesses. A 50 processor, 5 cluster Cm* machine was built and operated.

Cm* showed the feasibility of a bus architecture which was designed to take advantage of locality of reference by attaining linear speed-up on such clusterable applications as partial differential equations. The lack of cache memory simplified the role of the custom controller and eliminated the need for a cache coherence protocol. Amortizing the cost of the custom controller across the 10-14 processors in a cluster allowed it to be more powerful and expensive than if each processor were required to have its own controller. A buffered, packet switching bus protocol was used to avoid deadlocks over bus allocation. Not having to worry about a cache coherence problem made implementation of such a split-transaction protocol feasible.

Two ongoing research projects featuring hierarchical bus multiprocessor architectures have been reported in the literature. In a project known as the Ultramax [8, 56], Encore Computer, with DARPA funding, has made the logical extension to their Multimax architecture by joining single bus clusters of 16 processors with a global bus. The Ultramax uses large caches between the clusters and the global bus, and memory that is physically distributed among clusters.

The Data Diffusion Machine [29], from the Swedish Institute of Computer Science, is built around a multi-level hierarchical bus architecture. Processor caches at the lowest level of the hierarchy form the only system memory in the DDM. Caches at higher levels hold only status information, not data. The most novel aspect of the DDM is that shared memory data blocks are not considered to have a particular home location. Instead, data blocks migrate to where they are needed, living in caches at all times, without a home memory module for permanent residence.

1.6 Performance Evaluation

To be credible, a proposal for a new computer architecture must be accompanied by evidence of realistic requirements for and correct operation of system components. To be of interest, new computer architectures must show promise of increased performance. Two strategies were pursued to evaluate the HBSM multiprocessor architecture. Detailed Petri net models were used to design and debug the cache coherence protocol and controllers. Discrete event simulation of purely probabilistic and trace-driven models provided estimates of broad measures of performance of the protocol and the architecture.

1.6.1 Queuing networks

The main advantage of queueing networks is the balance between accuracy and efficiency which they provide. The close correspondence between certain aspects of computer systems and networks of queues eases the task of developing a queueing network model. Algorithms such as Mean Value Analysis are available for which the analysis of queueing network models is computationally proportional to the product of the number of queues and the number of customer classes. Use of such economical algorithms depends on being able to accurately represent the system under study with a class of models known as product form queueing networks. A further advantage is the fact that the number of parameters which must be obtained to characterize a system being modeled with product form queueing networks is relatively small. Queueing network models have been successfully applied to the analysis of high level computer system models where questions of capacity planning and overall system performance must be addressed.

If the system under study can not be adequately represented by product form networks, then a general network of queues can be employed. Unfortunately, analysis of such general networks is computationally very expensive; the amount of work required for analytical solutions rises exponentially with the size of the model. Simulation of queueing network models is an alternative in such cases, but is also computationally intensive and requires very careful characterization of the system under study to determine model parameters. For approximately the same investment in model development and simulation time, a more general discrete event simulation of the system under study can be made.

The main difficulty in using queueing networks for performance evaluation is that exact solutions to models exist only for product form, or separable, networks. Product form networks must be composed solely of first-come-first-served queues with exponential, class-independent service times. FCFS queues with exponential service times are not sufficient for accurately modeling the memory subsystem of a multiprocessor. As a result, approximate solutions to non-product form networks (those with a wider range of service times and disciplines) must be found. Sauer and Chandy [45] outline two approaches to this problem. The first is to simply apply the product form solutions to the non-product form model. In some cases, Reiser has shown that this approach can yield results close to the exact solutions [43]. The conditions under which this strategy yields results close to the exact solution are for queueing networks with large populations of customers. This requirement is not met for a model of the HBSM multiprocessor, since the number of customers in the entire model would simply be the number of processors being modeled. On average, most queues would never see more than one customer enqueued at a time.

The second approach to solution of non-product form networks is known as aggregation. While no known bounds exist for the error incurred by using aggregation, Lazowska, et al claim that a large body of empirical evidence exists to show that results from aggregation are usually close to the exact results [35]. The empirical evidence cited comes from the field of computer systems modeling and capacity planning. Unfortunately, the number of customers in a queueing network model of the HBSM multiprocessor would be so small, compared to the computer system models mentioned, that the error involved with aggregation is much more uncertain.

Queueing networks are unable to represent some common aspects of computer system behavior such as simultaneous holding of resources. The representation of

concurrency is an unfortunate limitation when using queuing networks to model parallel computer systems. Since queuing network models are concerned with average flows in a network of queues, it is difficult to represent resources which must be used exclusively. For example, consider a snooping cache controller competing with a processor for exclusive access to the cache directory. This competition (and the likely locking that accompanies it) has ramifications for the performance of the cache memory subsystem. In a queuing network model of such a cache controller, the use of the cache directory can only be represented as an average flow through the directory queue; competition and locking can not be directly represented.

Modeling with queuing networks is an entirely stochastic approach; the actual memory references recorded in address traces can not be used directly as input to a queuing network model. Increased availability of computer power at decreased cost has boosted the popularity of simulation as the performance evaluation tool of choice. More popular in the past, technological trends and the availability of sophisticated modeling and simulation software have largely displaced queuing networks as a performance evaluation tool for detailed computer architecture studies.

1.6.2 Discrete event simulation

The overall performance of a shared memory multiprocessor depends heavily on the performance of the memory hierarchy. In turn, the performance of the memory hierarchy depends on the memory reference patterns of an application. Interconnection network traffic attributable to the maintenance of cache coherence in particular is very sensitive to the distribution and timing of memory references both within a single processor and between processors. To be accurate, performance evaluation of a cache coherence system (protocol and controllers) must be done with trace-driven simulation. The difficulties of obtaining traces for multiprocessors provided the impetus for the development of a multiprocessor trace gathering technique that can be run on a uniprocessor. A major advantage of the traces generated in this manner are their architectural independence. Address traces are taken at the process level, so the traces are independent of the architecture of the machine generating them. Synchronization artifacts in the traces are avoided by recording the presence of barrier synchronization points in the traces themselves. The simulators that are fed with the traces determine the actual timing of the read, write, and synchronization events

recorded in the traces.

To demonstrate reasonable hardware requirements and to provide some degree of confidence in the correctness of the cache coherence protocol, detailed modeling was performed with high-level, timed Petri nets. The Petri net model accurately represents the complete state of the system controllers, buses, caches and memory modules. Developing this model and running it with multiprocessor address traces was crucial to the design and debugging of the proposed cache coherence protocol. However, the level of detail in the Petri net model precludes its use for trace-driven simulations of adequate length to accurately estimate the potential performance of the HBSM multiprocessor architecture.

Estimates of bus and controller utilization, the chief performance metrics of interest in bus-based multiprocessors, are made using an object-oriented, event scheduled, discrete event simulation developed in the C++ programming language. The C++ simulation is a hybrid approach combining trace-driven simulation and probabilistic models. For applications which are symmetric with respect to clusters, the cache activity on one cluster can be taken to be representative of the activity of all clusters. Simulation time is reduced by simulating the activities of only one cluster. The effect of the interaction between clusters is represented probabilistically in the simulation by pre-processing the address traces and estimating the amount of intercluster activity for each phase of the application.

1.7 Thesis Outline

The evolution of bus-based shared memory multiprocessors is depicted in Figure 1.2. The first step taken was to join several processors and a central memory to a single bus via private caches. This first step was motivated by the ever present need for increased computing power, the desirability of the shared memory programming paradigm, and the familiarity of computer engineers with bus technology. The scalability limit of the single bus multiprocessors is the bandwidth of the bus; this unique resource becomes the system bottleneck when the number of processors exceeds about 20. The graph in the middle of Figure 1.2 shows the behavior of speed-up or total compute power of single bus multiprocessors when the bus becomes a system bottleneck. Actual performance saturates at a limit determined by the bus bandwidth, the memory reference patterns, and the size, geometry and speed of caches and memory. Further

additions of processors at this saturation point does not result in further speedup. The natural next step to extend the scalability of bus-based multiprocessors is to form a hierarchy of buses, as shown in the bottom of Figure 1.2.

This thesis describes the design and performance of the hardware controllers and cache coherence protocol of a hierarchical bus shared memory multiprocessor. The first four chapters constitute the background of the problem and the paper design work. The final four chapters represent the work done to evaluate the HBSM multiprocessor architecture.

The cache coherence protocol and controllers developed for the hierarchical bus shared memory architecture are complex entities. Verification of the correctness of such complex, interrelated hardware/software systems remains an open research topic. Trace-driven simulation with an accurate and detailed Petri net model was performed to gain confidence in the correctness of the memory subsystem. The Petri net simulator used to develop and debug the protocol and hardware controllers is a powerful, general tool for the study and performance evaluation of concurrent systems. However, the level of detail of the Petri net model required to be useful for studying the correctness of the system precludes its use in performance evaluation of the system for trace files of realistic lengths. An object-oriented discrete event simulator was developed to study the higher level performance of the memory subsystem. Both strategies of this two-pronged approach, detailed Petri net models and higher level discrete event simulation, are trace-driven approaches. The goal of the first strategy is to gain confidence in the correctness of the protocol and the realizability of the hardware controllers. The goal of the second strategy is to demonstrate the feasibility of increasing the scalability of bus-based multiprocessors by an order of magnitude, from 10–20 processors to 100–200 processors.

1.7.1 Guide to chapters

This chapter outlines the need for parallel processing technology to bring to bear increased computational power on a wide range of computer applications. A brief review of the elements of parallel computer architectures and the method of applying caches to bridge the processor/memory speed gap was provided as background. The first cache-based single bus multiprocessors were described. The natural extension

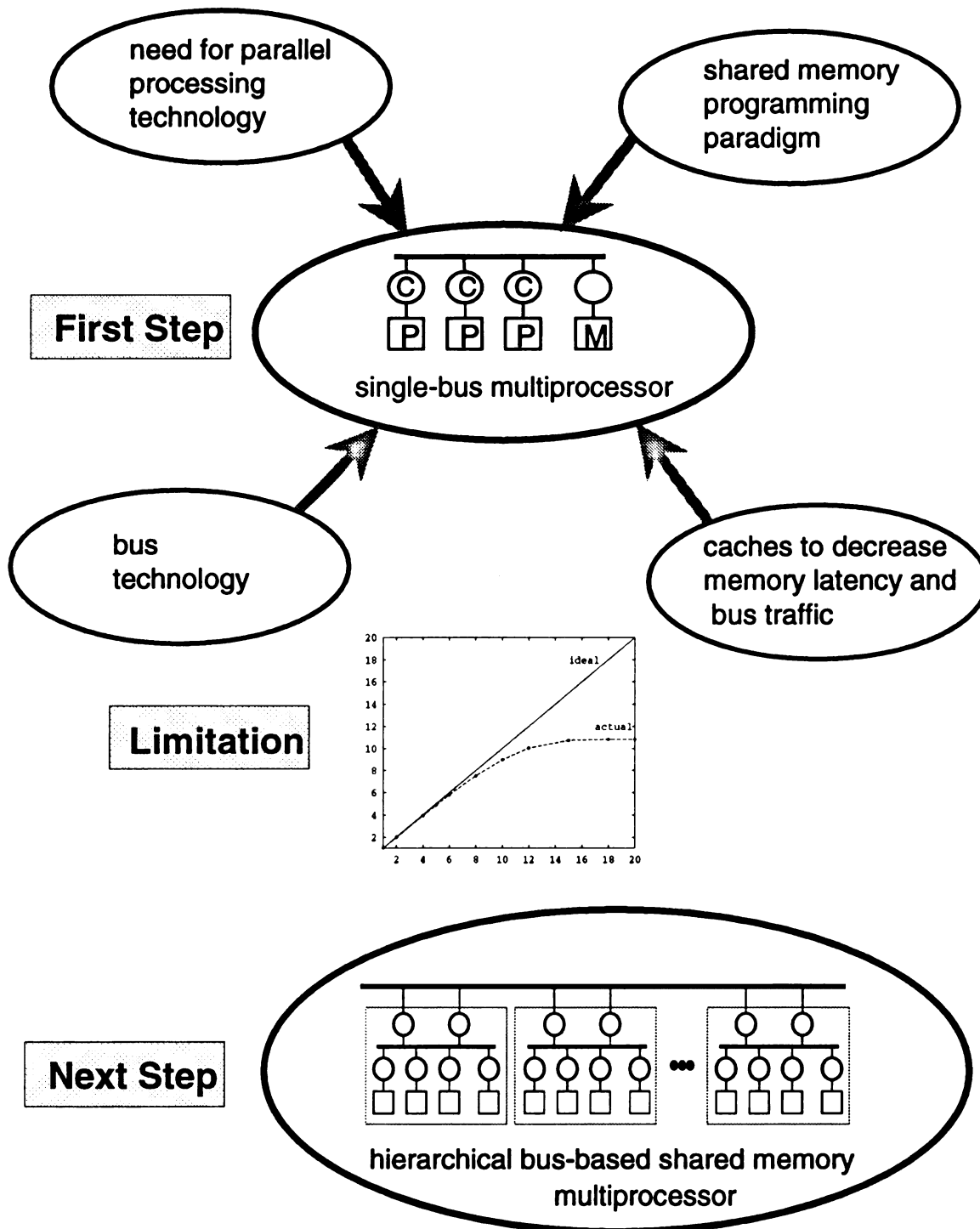


Figure 1.2: Steps in the evolution of bus-based shared memory multiprocessors.

of the hierarchical bus as a means of increasing the ultimate size of bus-based multiprocessors was outlined. Performance evaluation strategies were described which produce confidence in the correctness of the proposed cache coherence protocol and broad measures of performance for the hierarchical architecture.

The problem of cache coherence is considered in Chapter 2. Possible solutions to the cache coherence problem are described, with specific attention being paid to a class of hardware solutions for single bus multiprocessors.

The components of the hierarchical bus shared memory multiprocessor studied in this thesis are described in Chapter 3. The responsibilities and the design criterion of each controller are explained. The description in this chapter includes a section analyzing the hazards inherent in a controller which monitors two asynchronous buses and how each of these hazards is resolved.

The cache coherence protocol developed for the hierarchical bus multiprocessor is described in Chapter 4. A complete description of the states of the protocol, the required bus transactions, and the tables defining the protocol are found here.

Chapter 5 describes a method developed to generate multiprocessor address traces for detailed performance evaluation of multiprocessors. The chief advantages of this method are the lack of reliance on a real multiprocessor and the architectural independence of the address traces.

The detailed Petri net models used to design and debug the hardware controllers and the cache coherence protocol are discussed in Chapter 6. The models were used in a Petri net simulator developed for the purpose of studying concurrent systems. The simulator supports many extensions to Petri nets which increases the modeling power or expressiveness of the tool.

Discrete event simulation is used to estimate the performance of the hierarchical bus multiprocessor. The object-oriented simulation tools and the models used for performance evaluation are described in Chapter 7. A purely probabilistic model of the complete HBSM multiprocessor is used to explore the scalability of the hierarchical architecture. A trace-driven extension of this model is used to verify the purely probabilistic simulation model, and to study in more detail the effects of cache geometry and coherence traffic in the HBSM multiprocessor. This simulation model provides increased detail with acceptable speed by accurately capturing the state of a single cluster and representing intercluster activity probabilistically with parameters measured from address traces.

The final chapter draws conclusions about the scalability of the hierarchical bus architecture and considers directions for future research in several areas.

Chapter 2

Cache Coherence

A hierarchical bus multiprocessor has the potential of increasing the ultimate size of bus-based multiprocessors into the range of a few hundred processors. The critical role of caches in bus-based architectures and the obvious need for correct and deterministic operation of the multiprocessor means that a satisfactory solution to the cache coherence problem is necessary. As shall be seen, the performance of the system for maintaining cache coherence will be crucial to the performance of the overall system.

2.1 Definition of Cache Coherence Problem

The cache coherence problem arises from the simultaneous existence of multiple copies of the same block of memory. Physical memory of a computer can be thought of as a collection of memory blocks. Each block of memory is associated with a home memory module. Copies of each memory block may exist in zero or more caches in a multiprocessor. The correct and deterministic operation of the multiprocessor depends on managing the multiple copies of data blocks, or solving the cache coherence problem.

An intuitive requirement for a solution to the cache coherence problem is that any read to a memory location should return the value last written to that memory location. In a multiprocessor the meaning of *last written value* may very well depend on the point of view of each processor. As a result, this intuitive definition of coherence is not rigorous enough to use for evaluating solutions to the cache coherence problem, but does indicate a desired behavior for a solution.

A more rigorous requirement for correct operation is known as sequential consistency. Sequential consistency requires that a system produce the same result as long as the memory accesses of each processor are kept in order, while the memory accesses of different processors may be interleaved in arbitrary order. This is a common expectation among programmers, no doubt because uniprocessors execute in this fashion,

and the semantics of sequential and parallel programming languages encourage this model. Scheurich and Dubois define a sufficient condition for a multiprocessor to be sequentially consistent in terms of global performance of memory accesses. A store is globally performed when it is performed with respect to all processors, that is, no load issued by any processor can return an old value which was valid before the store. A load is globally performed when it is performed with respect to all processors, and the store which is the source of the value returned has been globally performed.

Sequential consistency is satisfied in *any* system if an access may not be performed with respect to any processor until the previous access by the same processor has been globally performed and if accesses of each individual processor are globally performed in program order [46].

Enforcing sequential consistency via the cache coherence system may be too costly. For example, if multiple caches in the hierarchical bus architecture held copies of the same block, sequential consistency would require that a processor writing to this block be held until the effect of the write operation had been recorded in every cache holding the block. Even if the effect of the write were to simply invalidate other copies of the shared block, stalling the processor until all invalidations were performed (and until acknowledgments had been received) would severely degrade performance.

Alternatives to sequential consistency are known as weak ordering models. Systems which do not enforce sequential consistency (or strong ordering) may still execute correctly if proper synchronization is followed. Coherence in a weakly ordered system is only enforced at necessary synchronization points. The advantage of weak ordering models is that by allowing inconsistency when it does not matter, the performance of the system may be improved. To continue the example used above, a processor would only need to be delayed on a write to a shared block until it broadcast its write intention on its cluster bus, knowing that invalidations would eventually be received by all caches holding a copy of the block.

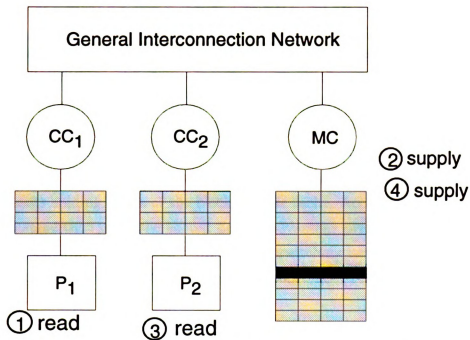
The cache coherence maintenance system (protocol and controllers) described in this thesis implements a form of weak ordering known as processor consistency. Processors are released after a write to a shared block once the transaction has been issued on the bus; a processor is free to issue further memory accesses before the write is seen by all processors in the system. Accesses by any given processor occur in sequential order with respect to that processor, but nothing is guaranteed about memory accesses relative to other processors. Not guaranteeing sequential consistency

eliminates the only published method of proving a cache coherence protocol correct [46]. The issue of correctness is addressed in later chapters. Processor consistency requires that access to shared variables be protected with synchronization constructs such as semaphores and monitors.

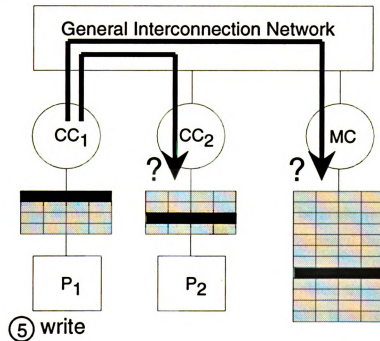
The expectations of a good solution to the cache coherence problem are that it not allow violations of the cache coherence protocol, that it maintain coherence with minimal impact on the performance of the system, that it be transparent to programmers, and that the cost and complexity of the solution are reasonable. A solution to the cache coherence problem does not eliminate the need for proper use of synchronization in parallel programming.

2.2 Example of Cache Coherence Problem

To understand the cache coherence problem, imagine a small multiprocessor with two processors each with private caches and a single system memory, joined by an arbitrary interconnection network, as shown in Figure 2.1a. The first event (step 1) in the scenario depicted is when processor P_1 does a read to a block which misses in the cache. The cache miss is satisfied by the cache controller CC_1 requesting that the block be supplied to it. The memory module responds to this request (step 2), and the block is loaded into the cache of P_1 . Processor P_2 proceeds to also read from this block (step 3), find it missing in the cache, load the block from memory (step 4), until the situation is as shown in Figure 2.1b. The next event in the system is a write by processor P_1 to the block. At this point, the question of updating the other copies of the block that exist in memory and cache arises. If the write to the block is passed over the interconnection network, thus keeping the copy of the block in memory up-to-date with the latest copy in P_1 's cache, then the cache coherence protocol is classified as *write-through*. If memory is left inconsistent with the latest cache copy, then the protocol is said to be *write-back*. A write-back protocol requires that caches take responsibility for blocks for which they hold the latest value. The actions taken for the copy of the block in P_2 's cache distinguishes the cache coherence protocol as either a *write-invalidate* or *write-update* protocol. Write-invalidate protocols simply invalidate copies of the block held in other caches. Write-update protocols transfer the new data value of the block to the other caches holding copies of the block. The memory update policy and the cache update policy are orthogonal.



a) Reads by P_1 and P_2 load copies of block into their respective caches.
Block of interest shown in black.



b) Write by P_1 forces choice of policy for updating memory and cache copies of block.

Figure 2.1: Simple multiprocessor scenario to demonstrate cache coherence problem.

2.3 Solutions to the Cache Coherence Problem

The strategies for solving the cache coherence problem can be broadly classified into hardware solutions or software solutions. The policies which can be implemented efficiently depend on the architecture of the multiprocessor, particularly of the interconnection network. For example, if all cache controllers can monitor broadcasts from all other cache controllers, then a write-update policy can be more efficiently implemented, since the writing processor need not know in advance the list of other processors holding copies of the block in their cache. Access to the same information at the same time gives the shared bus an advantage over more general forms of interconnection networks, such as multistage switching networks, since it allows for a distributed hardware solution to the cache coherence problem.

2.3.1 Software solutions

Software solutions to the cache coherence problem rely on intelligent compilers to generate instructions to the cache controllers which will ensure that coherence is maintained [12]. In the most extreme case, the compiler may simply mark all variables shared between two or more processors to be uncacheable. Accesses to share variables would bypass the cache and go directly to memory every time. A more sophisticated approach is to classify shared variable memory accesses by their coherence requirements, and issue instructions to the cache controller to maintain coherence for the shared variables but still allow caching. For example, shared variables that are used read-only by multiple processors can always be cached, and no special action need be taken to maintain coherence. Shared variables which are read by many processors and written by just one processor must be made uncacheable on the reading processors and can be cached for the writing processor if the cache protocol specified a write-through memory update policy.

Software solutions to the cache coherence problem do not require complex cache controllers, but do require that a processor be able to issue commands to the cache controller to turn the cache on and off, and to selectively invalidate cache blocks. Performance of software solutions can not be as high as a properly implemented hardware solution to the coherence problem, since compilers cannot know the relative timing between accesses to the same block by multiple processors. In other words, if

a block is accessed read/write by two processors, but those accesses do not overlap in time, then hardware support for cache coherence will allow greater cache accesses to occur with only small overhead, while a software solution enforces no caching of the block.

2.3.2 Hardware solutions

Placing the burden of cache coherence on hardware makes the actions necessary to maintain coherence transparent to both the programmer and the compiler writer. The solutions that have been proposed can be classified into directory schemes, snoopy controllers, and coherent networks. Directory schemes use state information held in either a centralized directory [54] or distributed among the memory modules [10]. In both cases, the directory holds state information which enables the cache controllers to take actions necessary for preserving coherence. Centralized directories may become a system bottleneck, as every memory reference must be checked against the directory. Stenström has proposed a directory scheme where the state information is distributed among the caches [50].

Full-map directories contain state information for every instance of a block in the system, allowing invalidates or updates to be sent only to those caches holding the block. The size of full-map directories has prompted methods of limiting how many caches can hold a block at the same time. These limited directories [11] either force invalidations when the requested number of blocks exceeds their capacity or resort to broadcasting invalidates and update information. The main advantage of directory schemes is they do not require a single source of information which all cache controllers can monitor; they can be used with general interconnection networks.

A second class of hardware solutions are the snooping schemes designed for single bus multiprocessors. When a single shared bus is used to interconnect all processors and memories, broadcast from one cache to all other caches is readily and inexpensively achieved. Snooping on (monitoring) the bus gives all cache controllers information about what is happening in the system simultaneously. A single invalidate bus transaction can invalidate all cached copies of the block at the same time. In addition, the bus acts to serialize accesses to memory, since the bus is the only path for all processors to memory, whereas in a generalized interconnection network there

may be multiple paths to memory which can be used concurrently. The characteristics which make the bus desirable from a standpoint of cache coherence act as a fundamental limitation to the ultimate size of bus-based multiprocessors. Because the bus is a unique resource shared by all processors to communicate with each other and memory, it will inevitably become the system bottleneck as the size (number of processors) of the multiprocessor is increased.

Limitations in the scalability of bus-based multiprocessors have led to proposals for cache coherent networks. These machines are multiprocessors built around interconnection networks designed to maintain cache coherence while at the same time not limiting the scalability of the overall system. Hierarchical bus architectures have been proposed by several researchers. Wilson [56] describes a two-level hierarchical bus machine which uses large second level caches to hold copies of the blocks held in caches in the cluster below them. These second level caches hold both status and data. The system memories are concentrated on the highest level (global) bus. The cache coherence protocol used is an extension of a simple invalidation protocol with write-back.

The Data Diffusion Machine [29] being built at the Swedish Institute of Computer Science uses large low-level caches attached to each processor and a hierarchy of buses connected by status-only caches at each hierarchical level. A novel feature of the DDM is the absence of traditional memory. The large caches at the processors act as a distributed memory for the entire system.

The Wisconsin Multicube [28] is based on a two dimensional grid of buses with memory modules for each column, and large caches at the intersection of rows and columns. Each cache controller snoops on both its column and row bus. A simple write-invalidate/write-back protocol is used together with the information of all modified blocks in all column caches held in each cache controller. A novel feature of the Multicube cache coherence scheme is that read requests can be handled by the closest cache containing a copy of the block, even if the block has not been modified. Normally system memory handles requests for blocks which have not been modified.

2.3.3 Single bus snoopy protocols

The protocol and cache coherence system developed in this thesis have their roots in single bus multiprocessor solutions to the cache coherence problem. Table 2.1

summarizes the characteristics of several of the important evolutionary steps made in snoopy cache coherence protocols.

Protocol	Memory policy	Cache policy	Comment
Write Once	dynamic	write-invalidate	first write to memory, subsequent writes to cache
Synapse	write-back	write-invalidate	first implemented snoop protocol, introduced concept of ownership
Dragon	write-back	write-update	used bus line to support cache-to-cache sharing

Table 2.1: Important stages in the evolution of snoopy cache coherence protocols.

Goodman's Write Once protocol [27] was the first write-invalidate protocol and also used a dynamic memory update policy. In Write Once, the first write to a cache block was passed on through to memory (write-through memory update policy) while subsequent writes to the same block were kept in the cache (write-back). The Synapse protocol [25] was used in the Synapse N+1 fault tolerant multiprocessor and represents the first implementation of a snooping protocol. The ownership concept, in which every block has a distinct and unique owner at all times, either memory or a cache controller, was introduced by Synapse. The ownership concept was taken the next logical step by the Dragon protocol [39], in which write-updates are performed to keep caches up-to-date with each other when a block is being written to by multiple caches. A cache is an owner of a modified block, and hence is responsible for supplying that block to read requests, until it either replaces the block and writes it back to memory, or until another cache writes to the block and assumes ownership. Dragon uses a bus line, called the Shared Line, for cache controllers to indicate whether a block is being shared among multiple caches. The shared line obviates the need to broadcast every write-update on the bus when it is known that no other caches have copies of the block. This effectively splits the modified state into two states, a modified-shared state, and a modified-private state.

Archibald and Baer [2] studied the performance of published snoopy protocols (including those in Table 2.1) for single bus multiprocessors with probabilistic workload models. Their results indicate that cache coherence protocols which allow dirty

sharing of blocks (true of Dragon) have significantly better performance, particularly when the degree of sharing is high, than do simpler invalidation protocols.

Chapter 3

Architecture of the HBSM Multiprocessor

Several strategies for solving the cache coherence problem were discussed in the previous chapter. The solution proposed for the hierarchical bus shared memory multiprocessor studied in this thesis is comprised of hardware controllers and a cache coherence protocol. This chapter describes the HBSM architecture and specifies the responsibilities of the hardware components of the memory system. The possible states of a memory block in each controller are described in anticipation of the definition of the cache coherence protocol in Chapter 4.

3.1 Introduction

The Hierarchical Bus Shared Memory (HBSM) multiprocessor consists of a two-level hierarchical bus interconnect with physically distributed memory and private processor caches. A cluster consists of multiple processors sharing a single cluster bus. Multiple clusters are joined by a single global bus. The interface between the global bus and the cluster buses consists of two devices: a memory controller and a status-only cache controller. Providing the illusion of a single, flat virtual address space in the presence of physically distributed memory is an important characteristic of the HBSM multiprocessor both for performance and ease of programming.

The main components of the HBSM multiprocessor are shown in Figure 3.1. Caches in the system are shown as boxes attached to controllers. Each row of the boxes corresponds to a memory block. Data in caches are gray, status information is white. Groups of blocks separated by thicker lines indicate set associative caches. Each cluster consists of Processors **P** with private caches **CC** and a dumb Memory Controller **MC** on a cluster bus joined to the global bus by the dual-bus snoop controllers, the Cluster Memory Controller **CMC** and the Cluster Cache Controller **CCC**. In order to support the illusion of a single shared address space, the **CMC** maintains

a directory of status information for every block in the cluster memory module. The CCC keeps a set-associative status-only cache containing an entry for every block held in a CC cache on the CCC's cluster. The CCC acts to maintain coherence between caches on different clusters.

The architecture of the HBSM multiprocessor is similar to other hierarchical bus multiprocessors. The HBSM architecture differs from the Ultramax by physically distributing the memory modules throughout the system, rather than concentrating them on the global bus [8]. Additionally, the role of the second level cache is different. The Ultramax second level caches are traditional caches that hold the both the data and status of memory blocks found in the processor caches below them. The HBSM multiprocessor CCC caches only the *status* of the memory blocks found in the lower level caches, and not their data values.

The paper design of new multiprocessors is a time honored tradition for doctoral degrees in the field of parallel processing. Novel methods of connecting processors with memories are not difficult to propose; whether the new architecture has any merit is more difficult to ascertain. As with any complex system, the performance of a large multiprocessor depends on the interaction of its many subsystems. To study all of these subsystems is both necessary for complete understanding of the whole, as well as impossible for anything but a large team of researchers. This chapter, along with Chapters 4 and 6 attempts to compromise between a purely academic exercise and a useful investigation of a practical multiprocessor architecture. Some aspects of the HBSM multiprocessor are not modeled, or are represented only implicitly. For example, the role of the operating system in a real multiprocessor cannot be understated, but is completely ignored here. The model used for processors is very simple.

The main components of the memory subsystem are described in this chapter; Chapter 6 is devoted to a detailed Petri net model of these controllers. The Petri net model captures the state of each controller and each memory block in the system. Trace driven simulation and the level of detail found in the models uncovered problems that would be faced in actually building a multiprocessor similar to the one studied here. The goal of the design portion of this effort was to confirm the practicality of the protocol and the memory subsystem controllers. For example, a protocol transaction which required instantaneous information available only on other clusters is not practical. Modeling the HBSM multiprocessor in detail has verified that each

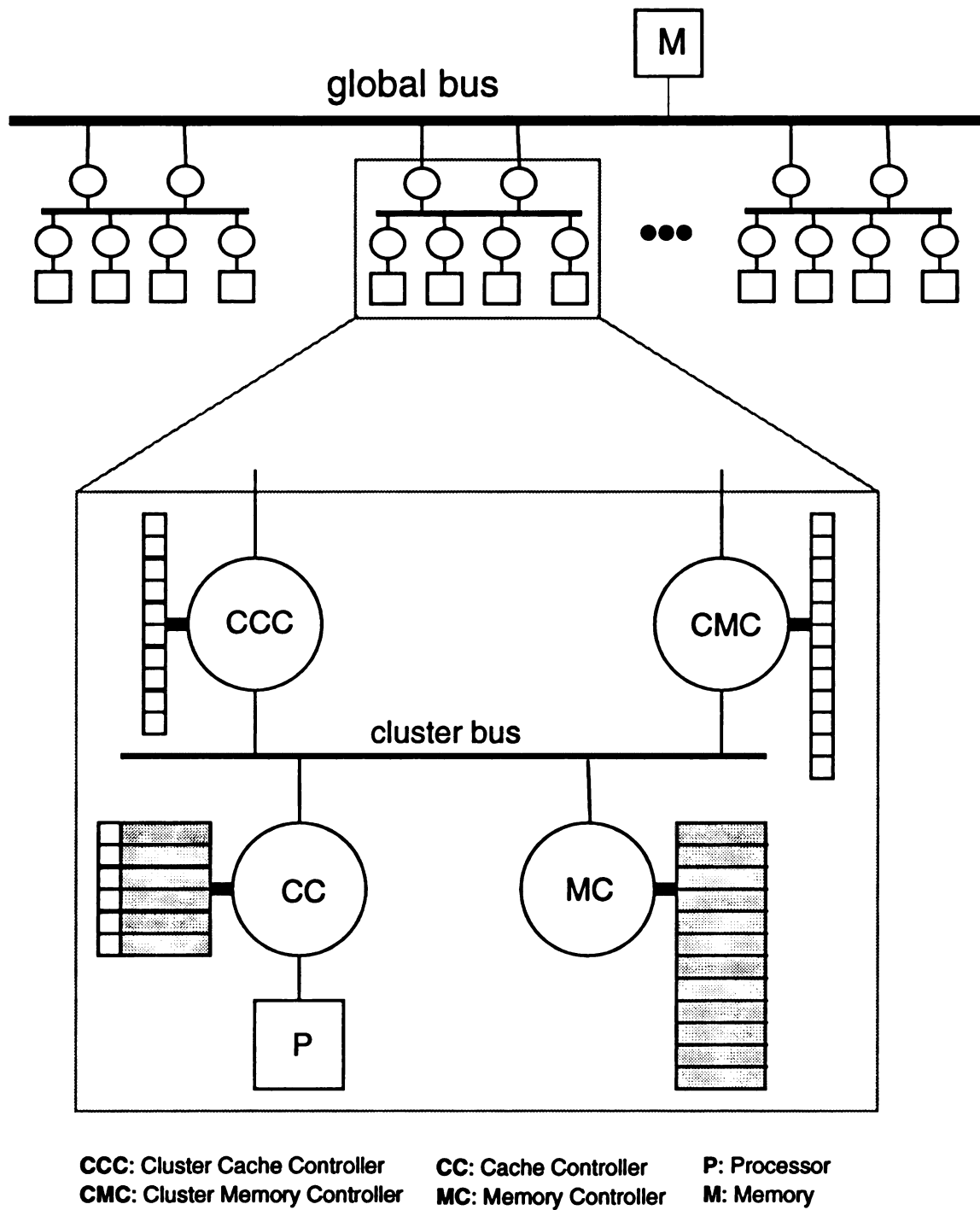


Figure 3.1: Components of the Hierarchical Bus Shared Memory (HBSM) multiprocessor architecture.

controller can act to maintain system coherence with information which would be readily available to it in a real system.

An additional achievement of detailed study of the cache coherence protocol and memory subsystem controllers is the knowledge of the phenomenon of state-space explosion which occurs in a complex, asynchronous system. Cache coherence protocols are often judged practical or impractical based on the number of states they utilize. What is not often discussed in research papers is the large increase in states and possible situations when the interactions of real controllers and asynchronous buses are considered. For example, when two processors on different clusters write to a common block at nearly the same time, the corresponding write notice transactions will collide at one of the dual-bus snoop controllers (CCC and CMC) What is represented by a single box in a protocol table has now become a complex event of aborting one of the transactions, resetting the states of various controllers and restarting the aborted cache write. If each of the many possible interactions on one cluster and between clusters is considered a state of the system, then the formal states of the cache coherence protocol mushroom into system states which must be properly handled.

A fundamental assumption underlying the design of the system controllers was the clustered nature of the architecture, and the expectation that computations performed on such a multiprocessor would exhibit clustered memory access patterns. The protocol and controllers were designed to be completely general, *i.e.* they will support any sort of memory access patterns. High performance is predicated on a bias of each processor to access memory locations found in their respective cluster memory module. The design objectives of the system were three-fold. First, the delay associated with intracluster memory accesses should be minimized since the system is aimed at applications with clusterable memory access patterns. Second, protocol traffic on the global bus should be minimized, since the global bus represents a possible system bottleneck. And third, the communication costs for intercluster accesses should be within an acceptable range to allow sharing of data amongst clusters.

3.2 System Controllers

The two dual-bus snoop devices, the Cluster Memory Controller CMC and the Cluster Cache Controller CCC each have two snooping parts; the cluster snoop (Csnoop) monitors a cluster bus, and the global snoop (Gsnoop) monitors the global bus.

Dual-ported memory could be used to implement a single directory to which both the Csnop and the Gsnop of these devices refer. Most of the complexity of the design of the HBSM memory controllers is found in the dual-bus snoop devices.

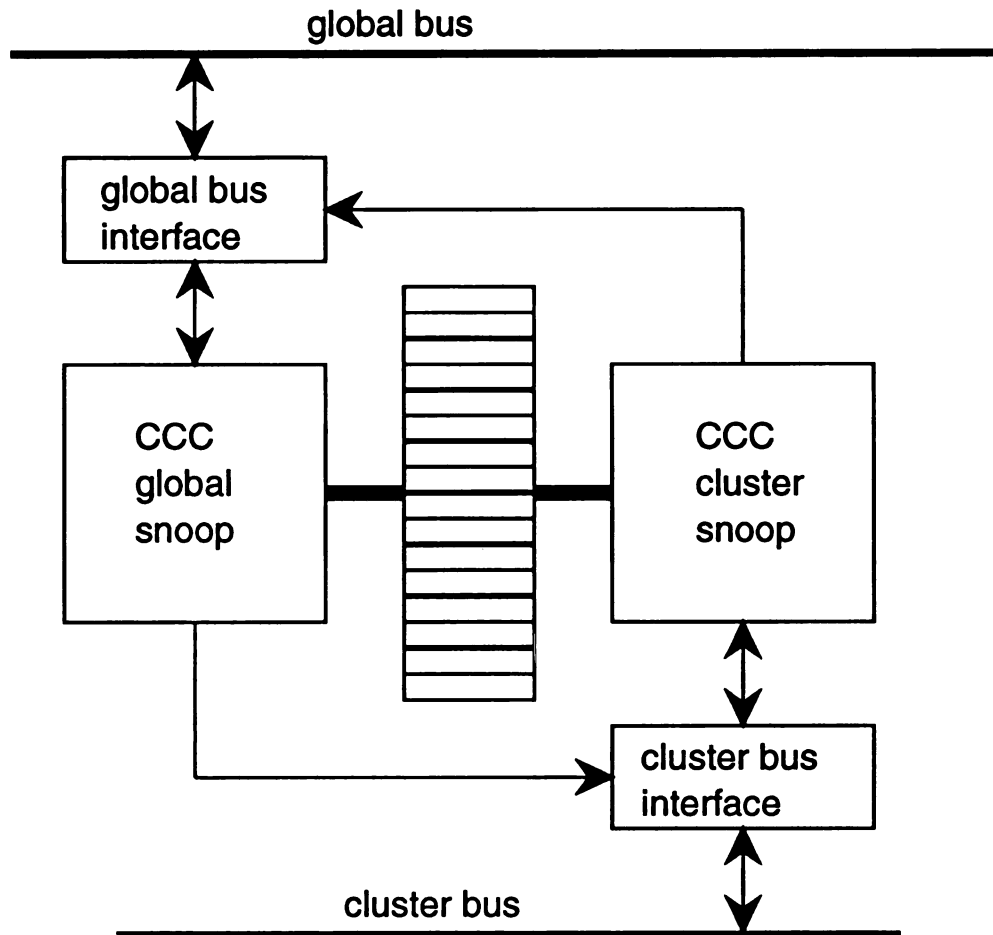


Figure 3.2: Detail of the CCC dual-bus snoop device showing the two active portions of the controller, cluster snoop and global snoop, sharing access to the single cache directory.

Explanation of a few key terms and concepts used will aid in understanding the HBSM multiprocessor and the cache coherence protocol. A *block* of memory corresponds to a contiguous group of memory locations, generally between 2 and 32 words. All blocks in the shared memory address space *reside* either in one of the cluster memory modules or in the global memory module. A block is *remote* with respect to a given cluster if it resides in the memory module of another cluster, or in global memory. A block is *local* with respect to a given cluster if it resides in the

memory module of that cluster. Similarly, clusters are said to be local or remote with respect to a given block. Devices will also be said to be local or remote with respect to another device based on whether the devices are on the same or different clusters. Caches *hold* copies of memory blocks. A cache block is the same thing as a cache line. Blocks held by caches on remote clusters are said to be *held remotely* and blocks held by local clusters are *held locally*. Copies of blocks can be either *unmodified* if the data in the cache and the data in memory are identical, or they may be *modified* if the data differ.

The MOESI [53] conventions for labeling the states are followed for the CCs. For the dual-bus snoop devices, the CCC and the CMC, the MOESI state names have been extended in a consistent fashion. Some states are said to be *intervenient*, which means that a cache with a block in such a state carries the responsibility for supplying the latest version of this block to other caches in the system. Cache controllers with blocks in intervenient states must perform a write-back of the block to memory if they wish to remove the block from their cache.

3.2.1 Cache Controller (CC)

Each CC has two independent components which utilize the same cache directory. The first component services requests from the processor and implements the cache replacement algorithm. The second snooping component monitors the cluster bus and takes coherence actions based on the contents of the cache directory and the bus transactions. Directory locking between components of the CC is done on a per-block basis. This scheme could be implemented in hardware with a dual-ported memory. Table 3.1 defines the states of blocks in the CC. Intervenient states Modified and Owned engender two responsibilities for the CC. First, read requests for such a block are satisfied by the snooping component of the CC. And second, such blocks must first be written back to memory if they are to be replaced in the cache. The CCC shares the responsibility for these intervenient states with the CC by monitoring requests on the global bus for these blocks.

3.2.2 Memory Controller (MC)

MCs are dumb devices which monitor their respective buses and respond to read and write requests for blocks within their memory modules. An MC can be inhibited from

CC State	Meaning
Invalid	block is invalid or not present
Shareable	block may be shared by other local or remote caches and be unmodified, or block may be modified and another local cache holds block Owned
Modified	block is modified and is held exclusively by this cache
Owned	block is modified but may be shared with other local caches

Table 3.1: The state of blocks in the CC.

responding to a bus transaction by a signal generated by a CC, CCC, or CMC.

3.2.3 Cluster Cache Controller (CCC)

The CCC monitors transactions on both the cluster and global bus and takes action according to the state of the referenced block as found in the CCC directory. The possible states of a block in the CCC are described in Table 3.2. Figure 3.3 illustrates the meaning of each state in terms of the fundamental attributes of a block from the perspective of the CCC. A block which is present on the cluster is held by at least one cache. Note that blocks in the Shareable Unmod and Cluster Exclusive states may have their state cached in the CCC while they in fact are not held by any caches on the cluster. This can occur when blocks that are held Shareable by CCs are replaced. The CCC has no way of determining when unmodified blocks have been removed from caches.

The main function of the CCC is to isolate the coherence traffic on the local cluster bus from the traffic on the global bus, while ensuring that coherence is maintained among clusters. The CCC maintains the status of every block held in the Cache Controllers of its local cluster bus. It is important to note that the CCC does not hold the actual cached data, but caches only the *status* of those blocks. CCCs also control the response of the global memory controller by preempting it from responding whenever one of their local caches has the most recent version of a global block. When a CCC must respond to a global bus transaction by taking some action on its local cluster bus it may preempt an existing cluster bus transaction which requires the

global bus, thus avoiding deadlock.

CCC State	Meaning
Invalid	no local cache holds a copy
Shareable Unmod	block is unmodified and either remote or local to this cluster, may be held Shareable by zero or more local caches; remote caches may also hold the block Shareable
Cluster Exclusive	local block is unmodified and exclusive to this cluster, may be held Shareable by zero or more local caches
Cluster Modified	block is modified and exclusive to this cluster; one of the local caches holds the block Modified or Owned

Table 3.2: The state of blocks in the CCC.

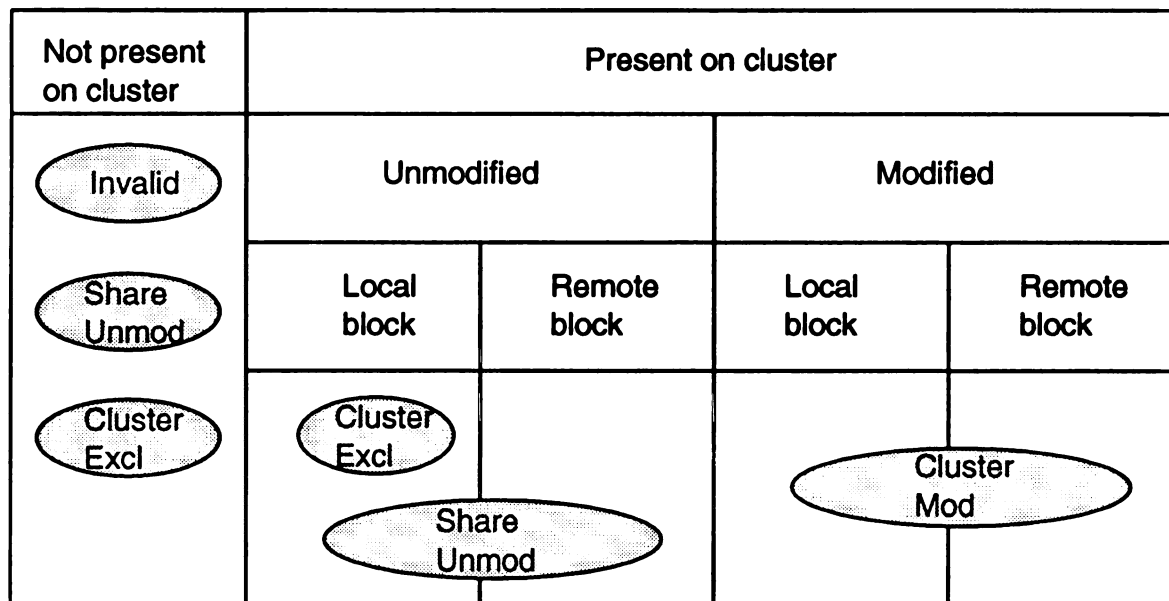


Figure 3.3: Meaning of the block states of the CCC.

Global bus read requests for blocks held in state Cluster Modified by the CCC are serviced by the CCC which issues a cluster bus flush for the modified block. CCC Gsnops are also responsible for translating global bus invalidates into cluster bus invalidates for blocks held Shareable Unmodified. On the cluster bus side, the CCC Csnop translates write notices into global bus invalidates for blocks that may be held in other clusters.

Since the CCC is organized as a set associative cache, and must hold the status of all blocks on the cluster, the CCC replacement policy will generate invalidate requests on the cluster bus when it must replace a block in its directory. As viewed by the CCs, these replacements are in addition to the replacements generated by the CCs own policy. When the CCC replaces a modified block, the CC which holds this block Modified or Owned is responsible for writing the block back to memory and then invalidating it. CCs holding the block Shareable simply invalidate the block. Read requests from either the global bus or the cluster bus for blocks with write backs pending in CCs must be intercepted by the CCC and aborted. Checking the queue of pending transactions against incoming transactions is one of the complexities that arises in the dual-bus snoop devices.

Selecting blocks to replace in the CCC would best be done in the following order of increasing cost in performance: Invalid, Cluster Exclusive, Shareable Unmodified, and Cluster Modified. In the worst possible case the CCC may select a Cluster Modified remote block to replace. The complexity of implementing an intelligent replacement policy must be compared to the cost in performance for a random replacement policy which is simple to implement.

3.2.4 Cluster Memory Controller (CMC)

The role of the CMC is to present the illusion of a single, virtual address space to the processors on its local cluster so that requests for a memory location can be made and serviced without knowledge of the resident cluster of the block. To do this, the CMC maintains state information for the blocks in the cluster memory of its local cluster; thus, the CMC always contains state information for the same blocks. This state information enables the CMC to prevent its local Memory Controller from responding to requests for invalid blocks. The CMC is also responsible for responding to local cluster read requests to remote memory modules. Table 3.3 gives the possible states of a block in the CMC. Figure 3.4 illustrates the meaning of each state in terms of the fundamental attributes of a block from the perspective of the CMC. Keep in mind that for the CMC, all states (except Remote, of course) are for local blocks. Thus for example, Invalid Locally could be more fully labeled Local-Invalid-Held Locally. The names have been shortened for ease of use.

To support the state of Cluster Exclusive in the CCC the CMC must at certain

CMC State	Meaning
Remote	block resides in a remote memory module; no state is kept in the CMC for remote blocks
Valid	no modified copies of the block exist, zero or more local or remote caches may hold copies
Cluster Exclusive	no modified copies of the block exist, zero or more local caches may hold copies
Invalid Locally	local cache has a modified copy of the block
Invalid Remotely	remote cache has a modified copy of the block

Table 3.3: The state of blocks in the CMC.

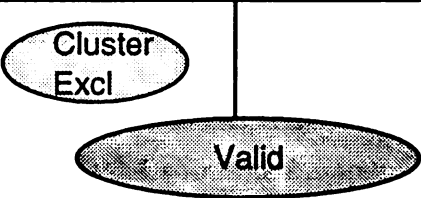


Remote blocks	Local blocks			
No state info kept	Memory is valid		Memory is invalid	
	Held locally	Held remotely	Held locally	Held remotely
				

Figure 3.4: Meaning of the block states of the CMC.

times inform the CCC of the exclusiveness of the block to that cluster. To perform this duty the CMC has to distinguish between those blocks checked out by remote caches and those held exclusively in local caches. This suggests distinct states of Valid and Cluster Exclusive in the CMC. Since blocks may be invalid in the cluster memory by being held modified either by a local cache or a remote cache, the invalid state of the CMC must be divided into two distinct states, called Invalid Locally and Invalid Remotely. For example, if a modified block is held by a remote cache (Invalid Remotely in the CMC), then upon receipt of a read request from a local cache, the CMC must relay the read request onto the global bus, while in the case of a local cache holding the block modified (Invalid Locally in the CMC), no actions by the CMC are required.

The Gsnoop side of the CMC responds only to global bus transactions for resident blocks; no actions are taken by the Gsnoop for remote blocks. Global bus read requests for clean resident blocks are translated by the Gsnoop into cluster bus read requests. Read requests for modified blocks are ignored by the CMC and are handled by the CCC. Global bus write backs to resident blocks are translated into cluster bus write backs by the CMC.

The Csnoop side of the CMC translates cluster bus requests for remote blocks into corresponding global bus requests. In addition, read requests for modified local blocks which are held remotely are serviced by the CMC. So that the other dual-bus snoop device, the CCC, does not need data paths to the cluster and global buses, the CMC acts in cooperation with the CCC on some transactions, relaying data from the cluster bus to the global bus. For example, a block held modified in a cluster is relayed to the global bus by the CMC Csnoop as a response to the CCC-generated cluster bus flush.

3.2.5 CCC/CMC state mapping

To help explain the relationship between the dual-bus snoop components of the HBSM architecture, Table 3.4 shows the mapping of states in the CMC to the CCC and CC for a given cluster. Table 3.5 shows the inverse mapping of states in the CCC to the CMC and CC for a given cluster. In these tables, AND, OR, NOT mean respectively the logical *and*, *or* and *not* operators. The + symbol means zero or more caches may hold the block in this state. The state of Invalid (absence of a block) is deleted for

clarity in the CC column. For each cluster the CC column represents the possible states of multiple CCs. For example, in the second row of Table 3.5, a block that is Cluster Mod in the CCC implies that the block is either held Modified by one CC, or that it is held Owned by one CC and Shareable by zero or more other CCs.

CMC state	CCC state	CC states
Remote	Share Unmod OR Inv OR Clus Mod	Share+ OR Mod OR (Owned AND Share+)
Valid	Share Unmod OR Invalid	Share+
Cluster Excl	Clus Excl OR Invalid	Share+
Inv Locally	Clus Mod	Mod OR (Owned AND Share+)
Inv Remotely	Invalid	

Table 3.4: Mapping of CMC states to CCC and CC.

CCC state	CMC state	CC states
Invalid	NOT Inv Locally	
Shar Unmod	Valid OR Remote	Share+
Cluster Excl	Clus Excl	Share+
Cluster Mod	Inv Locally OR Remote	Mod OR (Owned AND Share+)

Table 3.5: Mapping of CCC states to CMC and CC.

3.3 Interference Events

The asynchronous relationship between the global bus and the cluster buses produces a number of troublesome events which can occur within the dual-bus snoop devices. These events are referred to as *interference* events. Interference events are the nearly simultaneous presence or collision of transactions on the global bus and a cluster bus. These transactions interfere with each other at the interface of the those buses, that is to say, within a CMC or CCC.

Transactions which cause interference events can involve the same block or different blocks. Since both the Csnoops and Gsnoops of the dual-bus snoop devices access a single directory, a locking policy for these directories is required to reduce the possible combinations of same-block interference events that can occur. The locking of these directories is done by address on a per block basis; that is, simultaneous directory accesses are disallowed only for the same block. For different blocks, two accesses can proceed concurrently. The locking policy of the CCC and CMC directories reduces the number of possible same-block interference events that must be considered in the design of these devices to a manageable number by forcing the collision of each possible cluster bus transaction with each possible global bus transaction to occur in only two forms. In the first instance, the Csnoop gains a lock on the directory and the Gsnoop is locked out; in the second, the Gsnoop gains a lock on the directory and the Csnoop is locked out.

Since the directories of the dual-bus snoop devices are locked by block, interfering transactions which involve the same block will result in *directory deadlock*. Interference events resulting in directory deadlock can be classified as either *coherence events* or *benign events*. When the interfering transactions involve different blocks, we call the interference event a *collision event*.

3.3.1 Collision events

Collision events result in circular wait deadlock in the CCC and/or the CMC. When a dual-bus snoop device is required to translate a global bus transaction into a cluster bus transaction, the Gsnoop of the device issues an arbitration request to the current cluster bus master. The Gsnoop then waits to be granted the bus, allowing any ongoing cluster transaction to finish. The higher priority of the dual-bus snoop devices over the CCs ensures that the CMC/CCC Gsnoop will receive the bus on the next transaction. However, if the ongoing cluster bus transaction requires use of the global bus, then the Csnoop of either the CCC or the CMC will issue an arbitration request for the global bus. This situation is a circular wait deadlock, since the current global bus transaction can not finish without using the cluster bus, and the current cluster bus transaction can not finish without using the global bus. The Gsnoops of the CCC and CMC detect collision events by monitoring the global bus request queue of both the CCC and CMC Csnoops. Since collision events are different-block interference events,

they can not violate the coherence of the system, so the situation is resolved by giving the global bus transaction priority; the cluster bus transaction is aborted. This policy is consistent with the principle of aborting the transaction which originated latest in time and also decreases the traffic on the global bus.

3.3.2 Benign events

Benign interference events are those for which directory deadlock exists (because the competing transactions involve the same block) but the effects of the two transactions are compatible with each other and hence do not destroy the coherence of the system. These cases are handled simply by temporarily unlocking the locked directory, allowing the snoop which is in directory deadlock to continue its operations. The snoop which originally obtained the directory lock waits for the transaction on the other bus to finish, then relocks the directory while it continues its own transaction. As an example of a benign event, consider the following case. A read request occurs on the global bus for a block which is in state Valid in its CMC. The Gsnoop of the CMC locks the directory of the CMC. At nearly the same time, a cache on the same cluster as the CMC makes a cluster bus read request for this block. Since the CMC directory is locked, the cluster bus transaction can not be responded to by the Csnoop of the CMC. This benign interference event would result in permanent deadlock if the CMC were not capable of detecting the situation. The Gsnoop temporarily unlocks the CMC directory, allowing the cluster bus transaction to finish properly. The next transaction on the cluster bus is the CMC issuing a cluster bus read request to satisfy the pending global bus read request. Unlocking the directory temporarily is benign because there is no state change in the CMC for a read request to a Valid block.

3.3.3 Coherence events

Coherence events involve a cluster bus transaction and a global bus transaction which are mutually incompatible from a coherence standpoint. One of the two transactions must be aborted. As an example of a coherence event, consider the case when two caches on different clusters are sharing a block in an unmodified state. If both caches write to this block at nearly the same time, one of the CCCs of the active clusters will win arbitration and broadcast the global bus invalidate transaction before the other. In the second CCC (the one with the pending global bus invalidate transaction),

the directory will have been previously locked by the Csnoop for the cluster bus write notice transaction. Since these transactions are mutually incompatible with maintaining coherence between clusters, the write notice transaction on the the second CCC's cluster bus will be aborted.

Aborting transactions on the cluster or global buses poses difficulties in the design of the CCC and CMC. These devices must monitor the bus abort signal and be able to recover when an abort occurs. When a device is the perpetrator of an abort, it must either assume bus mastership for the next transaction, or allow the current bus master to pass mastership to one of the outstanding arbitration requesters. Devices which have initiated an aborted transaction should be persistent in re-trying the transaction.

3.3.4 CMC and interference events

The CMC is not responsible for generating transaction aborts during coherence and benign interference events; it does need to be able to react to them, however. CMCs which are the initiator of global bus read requests must re-try those transactions if the initial attempt ends in global bus abort. In addition, CMCs must be able to reset themselves at certain stages of reacting to an aborted transaction.

The CMC detects collision interference events by monitoring the global bus request queue of their own Csnoop and the CCC Csnoop. When a collision event is detected the cluster bus transaction is aborted.

3.3.5 CCC and interference events

The CCC is responsible for generating aborts for every coherence type interference event. An example of such an event is a global bus read request for a modified block and a cluster bus write back to that same block. In this case the CCC aborts the global bus read request, allowing the write back of the modified block to proceed. The two forms of this interference event, either the Csnoop locks the directory first or the Gsnoop locks the directory first, will be described in more detail to indicate the actions taken by the CCC and the problems that may arise in coherence events.

If the CCC Gsnoop decodes the global bus transaction and locks the directory first, the status of the block read by the Gsnoop when parsing the transaction will be Modified. Decoding the transaction puts a lock on the CCC directory. When the

Csnoop attempts to read the status of the block from the directory to respond to the cluster bus transaction deadlock will occur, since the Gsnoop holds the CCC directory lock. The Gsnoop detects this deadlock and resolves it in the following way. If the block is remote to this cluster, then a write back requires the global bus, and an abort of the global bus read request is mandatory. However, if the block is local to this cluster then the write back does not require the global bus, and it may seem that the global bus read request can simply wait for the write back to finish as in a benign interference event. The reason that this is not acceptable is that the Gsnoop has already locked the directory and issued a flush for the modified block. If the cluster bus write back were allowed to finish, the Gsnoop would then issue a flush for a clean block, which is a violation of the COGI protocol. Thus in both cases, for both local and remote blocks being written back, the Gsnoop must abort the global bus read request.

The other form this coherence event can take is the situation where the CCC Csnoop decodes its transaction and locks the directory first and the Gsnoop is locked out of the directory. For local blocks it would be possible for the write back to finish and then let the Gsnoop proceed in processing the global bus read request. However, write backs to remote blocks require use of the global bus, so the global bus read request must be aborted. Since the CCC performs its functions without distinguishing between local and remote blocks, both forms of this coherence event must be treated in the same way, and thus in both cases the global bus read request is aborted.

Table 3.6 shows the coherence interference events which are detected and managed by the CCC. To simplify the controllers, both aspects of the coherence events (Csnoop locks directory first or Gsnoop locks directory first) are handled in the same manner. Blocks replaced by the CCC may be the cause of coherence events. When the CCC forces a CC to purge a modified block (due to the CCCs replacement policy) the CC will hold this block in a pending write back state. Coherence events for blocks held in limbo awaiting replacement are marked with purge in Table 3.6. Read requests from either the global bus or the cluster bus for these blocks must be intercepted by the CCC and aborted. Other transactions may intervene on the cluster bus between the invalidate transaction and the CCs write back of the modified block.

Cluster trans	Global trans	CCC state	Action
write back	read request	Cluster Modified	abort global trans
write notice	read request	Shareable Unmod	abort global trans
write notice	invalidate	Shareable Unmod	abort cluster trans
read request	invalidate	Shareable Unmod	abort cluster trans
read request	-	Cluster Modified (purge)	abort cluster trans
-	read request	Cluster Modified (purge)	abort global trans

Table 3.6: Coherence events detected by the CCC and their resolution.

3.4 Bus Arbitration

Bus arbitration is distributed amongst potential bus masters. The current bus master advertises for arbitration requests and chooses the bus master elect for the next transaction. A bus master retains mastership, and hence responsibility for arbitration, until the bus is handed off to another bus master. Once selected, a bus master elect can be replaced only by a bus abort with bus master preempt. On a cluster bus, CCs have equal arbitration priority, and fairness is achieved among competing CCs by preventing a CC who has used the bus from participating in arbitration again until all outstanding requests have been satisfied. The priority of CCCs and CMCs is higher than CCs to minimize the amount of time the global bus is held by these devices.

If no requests are received by the arbiter during arbitration for the next bus mastership, the CCs which are locked for fairness are released and again allowed to participate in arbitration. CCCs and CMC do not have fairness restrictions for bus arbitration. Since the global bus is the most valuable resource of the entire system, the CCCs and CMCs may request the cluster bus at any time. On the global bus, CCC Gsnoops and CMC Gsnoops compete equally with a fair arbitration locking scheme identical to that for CCs on the cluster bus.

Chapter 4

COGI Cache Coherence Protocol

The definition of the Cluster Ownership Global Invalidation (COGI) cache coherence protocol is contained in the tables of this chapter. Using tables to define the protocol is a more structured method of displaying information which is often described using state transition graphs. The meaning of the protocol states for each of the active controllers is found in Chapter 3. For a more intuitive understanding of the COGI protocol, this chapter includes a lengthy example of intercluster sharing transactions.

4.1 Design of COGI Protocol

The Cluster Ownership Global Invalidation (COGI) cache coherence protocol has been designed to match the architecture of the HBSM multiprocessor. The COGI protocol is compatible with the MOESI class of cache coherence protocols [53]. On the cluster buses, COGI is a write-back/write-update protocol that allows dirty sharing in a manner similar to the Dragon protocol [39]. A bus line is used so that caches can know when other caches share a block. This allows two modified states to exist, one in which a cache has an exclusive copy of a block, and hence may write into the block without notifying other caches, and another where a cache is sharing a modified copy of a block with other caches and hence must broadcast updates to this block when it writes into it. These broadcasts are known as write notices and do not update memory, allowing for efficient sharing of data within a cluster. Blocks may be “checked out” from memory for writing, and shared among processors without being written back to memory until replaced. This efficient intracluster sharing is important, since in the applications suitable to the HBSM multiprocessor, the majority of processor memory accesses are destined for local memory modules. A block is never present in a modified state in more than a single cluster at a time.

Unlike Dragon, in the COGI protocol caches must broadcast their first write into a block. This write informs the second level cache controller (CCC) of the modified nature of the block. As a result of this requirement, the COGI protocol cannot use

a “valid exclusive” state for the processor caches. In Dragon, the valid exclusive state allows caches to write to blocks without any bus traffic; a valid exclusive block which is written moves to the dirty state, indicating the cached copy is modified and is exclusive. After the first write to a block the COGI protocol performs as well as Dragon.

The global bus protocol of COGI is a simple write-back/write-invalidate type. To reduce traffic on the global bus intercluster write sharing of data is limited to a series of read/invalidate/write actions. A cache secures an unmodified copy of a block with a read request. When this cache first writes to the block, the CCC invalidates copies of the block on remote clusters, thus avoiding the need for the broadcast of multiple write notices on the global bus. Although this decision discourages sharing of modified copies among clusters, it has no effect on sharing of unmodified blocks. In addition, CCCs are able to recognize local blocks held exclusively on their own cluster to avoid the necessity of broadcasting global bus invalidations for blocks not shared with other clusters.

The abbreviations used for bus transactions and the general meaning of these transactions are given in Tables 4.1 and 4.2. The various types of bus transactions are listed, and common (though not exhaustive) examples of their use are given.

4.2 COGI Protocol Tables

Tables 4.3–4.9 define the COGI protocol. Each row of the tables represents a state that a block can be in in that particular device. The columns represent various bus transactions, processor requests, and replacement actions. The tables are used by finding the intersection of the column corresponding to the transaction of interest with the current state of the block. Each cell of the table shows the actions taken by the device for this transaction in the top part, while the bottom part gives the change in state of the block. The notation “sigline ? *action*₁ : *action*₂” is used to indicate that ‘*action*₁’ is taken if ‘sigline’ is raised and ‘*action*₂’ is taken otherwise, where ‘sigline’ can be either of the two signal lines CSHL or REML. The notation \square signifies that the related transactions can not occur for a block in that state. The phrase “Read block cbus” and “Read block gbus” means that the controller copies a block off the respective bus. The phrase “Write block cbus” and “Write block gbus” means that the controller writes a block to the respective bus as the response to a

Abbreviation	Transaction	Example
CBRR	Cluster Bus Read Request	generated by cache misses
CBWN	Cluster Bus Write Notice	issued to update caches which are sharing a modified block
CBWB	Cluster Bus Write Back	generated by processors replacing a modified block
CBIN	Cluster Bus INvalidate	issued by the CCC as a result of either a CCC replacement or a GBIN transaction
CBFL	Cluster Bus FLush	used by the CCC to force the cache holding a modified block to provide that block for a GBRR
PR	Processor Read	read one word of a block
PR	Processor Write	write one word of a block
Purge	block replacement	block is selected for replacement by CC

Table 4.1: Transaction types on the cluster bus and processor events.

Abbreviation	Transaction	Example
GBRR	Global Bus Read Request	occur when a cache controller issues a read for a block which is owned by a device not on the same cluster
GBWB	Global Bus Write Back	occur when a cache controller replaces a block remote to its cluster that has been held modified
GBIN	Global Bus INvalidate	occur when a cache controller issues a write to a block that had previously been shared between clusters

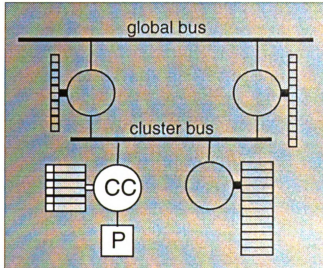
Table 4.2: Transaction types on the global bus.

bus transaction request. The phrase "Read word" means that the controller copies a single word from the bus to update a cache block.

The signal lines required on cluster buses are the CSHL (Cluster SHared Line) and REML (REMOte Line). Protocols based on write broadcast require an extra signal line, called the Shared Line, for the exchange of information regarding the shared status of blocks. Since the CCCs can not distinguish the locality or remoteness of addresses, the CMC use a second bus line, called REML, to inform them of the local or remote nature of blocks in a read request.

	Purge	PR	PW
Invalid		CBRR Read block cbus	
		→Shareable	
Shareable	Purge		CBWN
			CSHL?→Owned:→Modified
Modified	CBWB Purge		
Owned	CBWB Purge		CBWN
			CSHL?—:→Modified

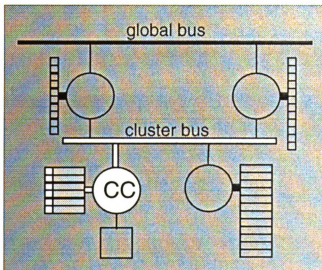
Table 4.3: Actions of CC in response to processor requests.



Processors make memory access requests (loads/stores or reads/writes) to the cache controllers. The cache controller services the memory access request from the cache if possible (cache hit) and by making a bus transaction if not possible (cache miss). All blocks are loaded into the cache on a cache miss in the Shareable state. Writes to a Shareable block result in the block changing to a dirty state, either Owned or Modified, depending on whether other caches in this cluster have copies of the block. The Purge column shown in the table above correspond to actions taken by the cache controller, CC, when a block is selected for replacement.

	CBRR	CBWN	CBWB	CBIN	CBFL
Shareable		Read word Raise CSHL	Raise CSHL		
				→Invalid	
Modified	Inhibit memory Write block cbus Raise CSHL †	<input type="checkbox"/>	<input type="checkbox"/>	CBWB	Write block cbus
	→Owned	<input type="checkbox"/>	<input type="checkbox"/>	→Invalid	→Shareable
Owned	Inhibit memory Write block cbus Raise CSHL	Read word Raise CSHL	<input type="checkbox"/>	CBWB	Write block cbus
		→Shareable	<input type="checkbox"/>	→Invalid	→Shareable

Table 4.4: Actions of CC in response to cluster bus transactions.

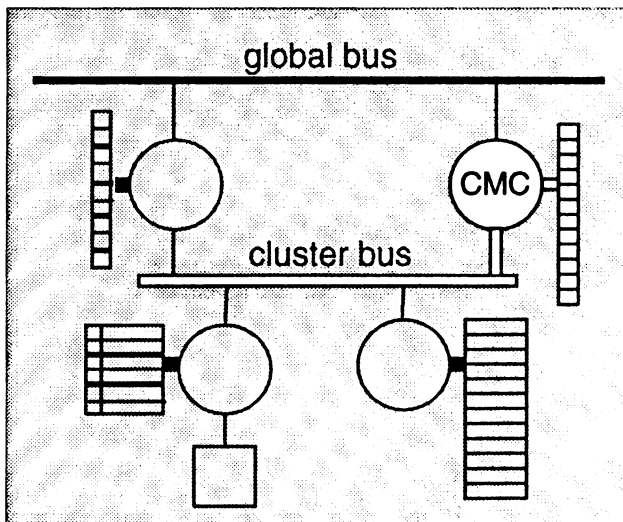


The other responsibility of the CC cache controller is to monitor transactions on the cluster bus and take actions to preserve cache coherence. The CC is responsible for supplying dirty blocks to cluster bus read requests, and also for updating copies of blocks that are being dirty-shared with other caches. The cluster bus invalidate and cluster bus flush transactions are generated by the cluster cache controller CCC. A CC must respond to these transactions by performing a write back of the block, and by invalidating the block, respectively.

†The raised CSHL notifies the CMC that the remote block is being supplied by a local CC and there is no need for a global bus read request.

	CBRR	CBWN	CBWB	CBIN	CBFL
Remote	CSHL?—: (GBRR Read block gbus Write block cbus Raise CSHL)		Read block cbus GBWB		Read block cbus Write block gbus
Inv Locally					Read block cbus Write block gbus
			→Cluster Excl		→Valid
Inv Remotely	Inhibit memory GBRR Read block gbus Write block cbus Raise CSHL	<input type="checkbox"/>	<input type="checkbox"/>	†	<input type="checkbox"/>
	→Valid	<input type="checkbox"/>	<input type="checkbox"/>		<input type="checkbox"/>
Valid	Raise CSHL		<input type="checkbox"/>		<input type="checkbox"/>
		→Inv Loc	<input type="checkbox"/>		<input type="checkbox"/>
Cluster Excl			<input type="checkbox"/>		<input type="checkbox"/>
		→Inv Loc	<input type="checkbox"/>		<input type="checkbox"/>

Table 4.5: Actions of CMC in response to cluster bus transactions.

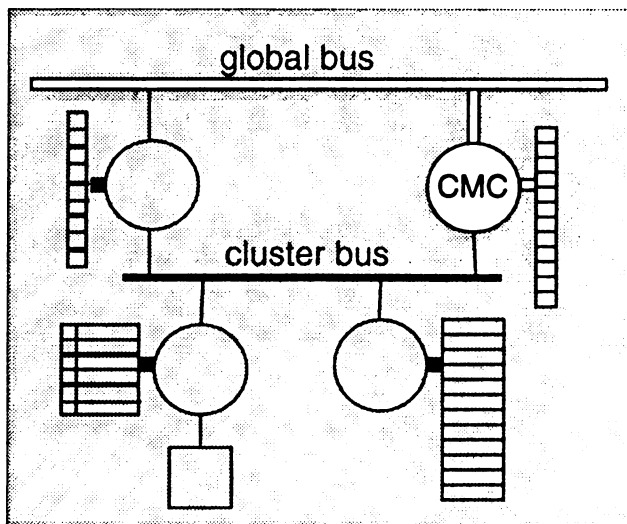


The CMC maintains the transparency of a physically distributed memory. On the cluster bus, it is responsible for servicing read requests for remote blocks and for local blocks that are held dirty on some remote cluster. The CMC acts to inhibit the MC from responding for a block by marking blocks that have been modified as invalid. Read requests for Invalid blocks are serviced by either a cache with a dirty copy (Invalid Locally) or by the CMC (Invalid Remotely). In either case the CMC inhibits the MC from responding to the read request.

†This scenario happens when a remote CCC is the originating source of a GBIN transaction. The local CCC has the block in state Valid, observes the GBIN and generates a CBIN. In the meantime, the local CMC Gsnoop observes the GBIN and changes state to Invalid Remotely. The CMC cluster snoop observes the CBIN transaction when it appears on the cluster bus, but the block is held in state Invalid Remotely this time.

	GBRR	GBWB	GBIN
Inv Locally		<input type="checkbox"/>	†
		<input type="checkbox"/>	
Inv Remotely	Read block gbus CBWB	Read block gbus CBWB	<input type="checkbox"/>
	→Valid	→Valid	<input type="checkbox"/>
Valid	CBRR Raise REML Read block cbus Write block gbus	<input type="checkbox"/>	
		<input type="checkbox"/>	→Inv Rem
Cluster Excl	CBRR Raise REML Read block cbus Write block gbus	<input type="checkbox"/>	<input type="checkbox"/>
	→Valid	<input type="checkbox"/>	<input type="checkbox"/>

Table 4.6: Actions of CMC in response to global bus transactions.

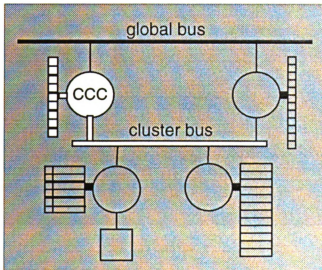


The global snoop of the CMC monitors the global bus and responds to requests for blocks belonging to the local cluster memory. The CMC translates global bus read and write back requests into cluster bus read and write back requests.

†This scenario can happen because the local CCC might be the originating source of the global bus transaction. The local CCC observes a CBWN for a local block in state Valid and generates a GBIN. In the meantime, the CMC cluster snoop observes the CBWN as well and changes the block state to Invalid Locally. The CMC Gsnoop observes the GBIN transaction when it appears on the global bus, but the block is held Invalid Locally this time.

	CBRR	CBWN	CBWB	Purge
Invalid		<input type="checkbox"/>		
	REML?—: (CSHL? →Shar Unmod: →Cluster Excl)	<input type="checkbox"/>		
Shar Unmod		GBIN	<input type="checkbox"/>	CBIN Purge
		→Cluster Mod	<input type="checkbox"/>	
Cluster Excl	†		<input type="checkbox"/>	CBIN Purge
		→Cluster Mod	<input type="checkbox"/>	
Cluster Mod				CBIN Purge
			CSHL?→Shar Unmod:→Invalid	

Table 4.7: Actions of CCC in response to cluster bus transactions.

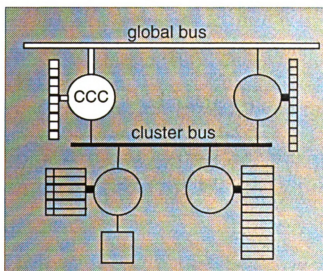


The cluster snoop of the CCC translates cluster bus write notices into global bus invalidates for blocks that may be shared by other clusters. After a first write to a Shared Unmodified block, the CCC does not send global bus invalidates, since it knows the block no longer exists in other clusters. During a cluster bus write back, the CCC will drop a block (set its state to Invalid) if no caches respond with the shared line. In general, however, the CCC has no way of knowing when caches drop blocks (via replacement), thus the CCC may cache copies of blocks which no longer exist on the cluster. These blocks will eventually be replaced by new, active blocks.

[†]This scenario starts with a read request on the global bus for a local block which is held in the Cluster Exclusive state of the CCC. The CCC Gsnoop observes this transaction on the global bus and changes the block state to Shareable Unmodified. The CMC Gsnoop relays the global bus request onto the cluster bus. When the local CCC cluster snoop observes the corresponding CBRR, the block is held Shareable Unmodified. Another scenario that is possible is a CBRR for the local block. In this case too, no state change is required.

	GBRR	GBWB	GBIN
Invalid			
Shar Unmod		†	CBIN
			→Invalid
Cluster Excl		<input type="checkbox"/>	<input type="checkbox"/>
	→Shar Unmod	<input type="checkbox"/>	<input type="checkbox"/>
Cluster Mod	Inhibit global memory CBFL	<input type="checkbox"/>	<input type="checkbox"/>
	→Shar Unmod	<input type="checkbox"/>	<input type="checkbox"/>

Table 4.8: Actions of CCC in response to global bus transactions.

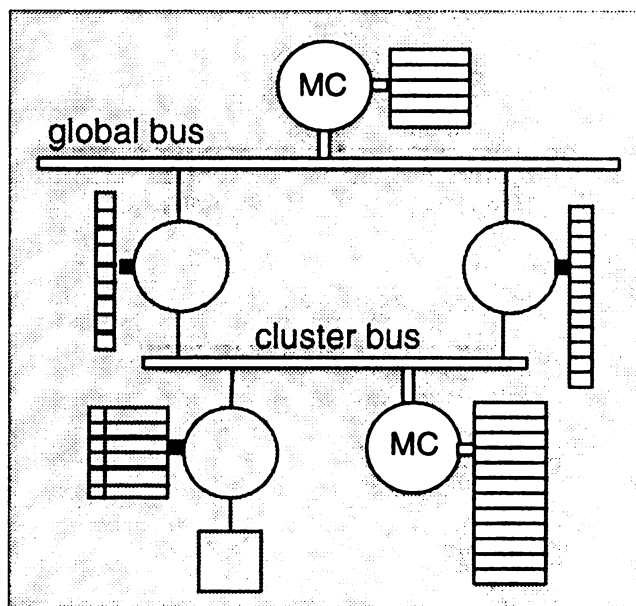


On the global bus, the CCC translates invalidate transactions into cluster bus invalidates for blocks held on its cluster. Additionally, the CCC Gsnoop must force CCs to supply dirty blocks by flushing them back to memory. When a global bus read request to a dirty global block is made, the CCC must also inhibit the global memory module from responding.

[†]This scenario starts with a write back of a remote block held modified by a CC. Assuming that there are still caches who have the block in Shareable, the CCC cluster snoop changes the state from Cluster Modified to Shareable Unmodified. The CMC relays the write back over to the global bus by generating a GBWB. The CCC Gsnoop then observes a GBWB to a block in the Shareable Unmod state.

	xxRR[†]	CBWN	xxIN[†]	xxWB[†]	CBFL
Remote					
Local	MC Inhibit?—: (Read block from memory, Write block onto bus)			Read block from bus, Write block into memory	Read block from bus, Write block into memory

Table 4.9: Actions of MC in response to bus transactions.



Memory controllers are very simple devices. They monitor their respective buses for transactions that correspond to blocks they own, and unless inhibited by another device on the bus, act to service the transaction. They carry no state information for the blocks they own.

[†]xx can be either of the two notations: CB or GB.

4.3 Example of COGI Protocol

The following example of the operation of the protocol involves intercluster cache references and is shown in Figures 4.1 and 4.2. The numbered steps in the figures follow the description of the events in the following text. A cache miss on a $P_{1,1}$ read request causes $CC_{1,1}$ to broadcast a read request (CBRR) over Cluster Bus 1 (see Figure 4.1). Assuming that the block is resident locally and no other cache has a copy, MC_1 responds to the request and provides the block. $CC_{1,1}$ reads the block off the cluster bus and sets its state to Shareable. CCC_1 observes the read request on the cluster bus and checks the REML to make sure that the request is not generated remotely and relayed by CMC_1 from the global bus. Next it watches for CMC_1 to raise the CSHL line to see whether the block is already being shared by other clusters. In the absence of both of these signals CCC_1 creates space for the block state and sets it to Cluster Exclusive, meaning that it is an unmodified local block and no remote cache has a copy.

A subsequent read request by $P_{1,10}$ and a CBRR broadcast over the cluster bus by $CC_{1,10}$ is answered again by MC_1 . $CC_{1,10}$ loads the block off the bus and sets its state to Shareable. As the next event of interest, $P_{1,10}$ writes into the block causing $CC_{1,10}$ to broadcast the modified portion of the block over the cluster bus as a CBWN and to change the state to Owned. $CC_{1,1}$ upon observing the write notice transaction updates that particular word in its copy of the block, but keeps the block in the state Shareable and raises the CSHL telling $CC_{1,10}$ to retain the block Owned. CCC_1 upon observing the modification to the block changes the state of the block to Cluster Modified while CMC_1 changes it to Invalid Locally, indicating that some local cache has a modified copy of the block.

At this point, the two caches $CC_{1,1}$ and $CC_{1,10}$ hold the block Shareable, and Owned, respectively. The CCC on Cluster 1 holds the block status as Cluster Modified, indicating a modified version of the block is exclusive to this cluster. To introduce some intercluster sharing, consider what happens when $P_{2,1}$ performs a read causing a read miss in its cache (see Figure 4.2).³ To get the block, $CC_{2,1}$ generates a read request over Cluster Bus 2. Since the block is remote to this cluster, CMC_2 monitors the CSHL to make sure that no local cache is responding to the read request. Since the block is not held modified by any of the caches on Cluster 2, CMC_2 broadcasts

³Due to the space limitation in Figure 4.2, the global bus is shown in two sections.

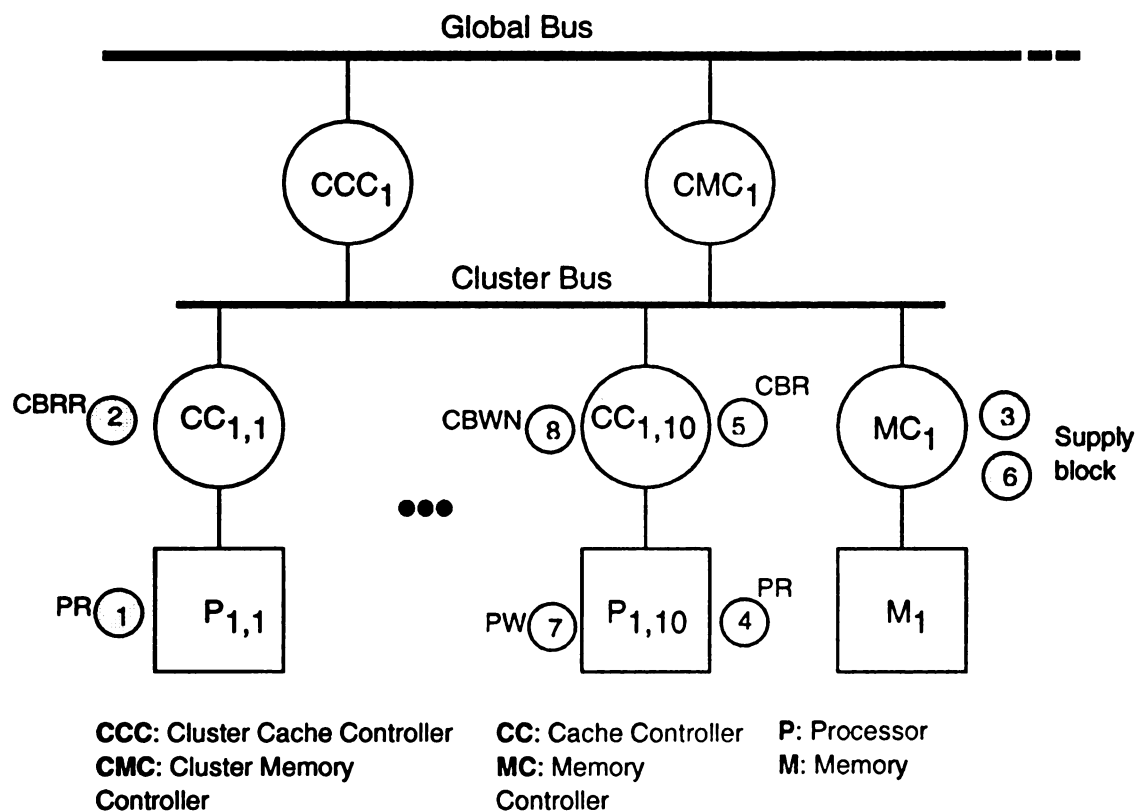


Figure 4.1: Basic scenario of intracluster sharing of locally resident block. Processor P_1 reads block, P_{10} reads then writes block, passing updated version of data word to P_1 .

a read request over the global bus. CCC_1 , having the block in the state Cluster Modified, generates a flush request (CBFL) on Cluster Bus 1 when it observes the global bus read request.

CCC_1 changes the state of the block to Shareable Unmodified indicating that the block is unmodified and shared between its caches and the remote cache on Cluster 2. Because $CC_{1,10}$ has the block in state Owned it responds to the flush request and provides the modified copy of the block on the cluster bus. $CC_{1,1}$ does not react to the flush request because the Shareable state is not intervenient, as is the Owned state. When MC_1 observes the flush it updates the copy in memory, making the block now unmodified. Acting in concert with CCC_1 , CMC_1 recognizes the global bus read request for a block whose state is Invalid Locally, and relays the block from the cluster bus to the global bus.

CMC_2 reads the block off the global bus as the response to its read request and places the block on Cluster Bus 2. It also raises the CSHL to signal to CCC_2 that the block is resident remotely and might be shared with caches on other clusters. CCC_2 , upon observing the CSHL signal from CMC_2 indicating that the block is shared with other clusters and/or owned remotely sets the status of the block to Shareable Unmodified. At the same time $CC_{2,1}$ reads the block off the cluster bus and sets its state to Shareable.

At this point the block exists in three caches, $CC_{1,1}$, $CC_{1,10}$ and $CC_{2,1}$ and is in the Shareable state in all three. Now consider a write to the block by $P_{2,1}$. $CC_{2,1}$ broadcasts the written word over the cluster bus (CBWN), and monitors the CSHL line. Due to a lack of response on the CSHL line, $CC_{2,1}$ changes the state of the block to Modified, indicating that no other local caches are sharing the block. When CCC_2 observes the write notice on the cluster bus it broadcasts an invalidation transaction (GBIN) on the global bus and changes its state to Cluster Modified. CCC_1 observes the invalidate transaction, relays it to Cluster Bus 1 as a CBIN transaction, and sets the state of the block to Invalid. $CC_{1,1}$ and $CC_{1,10}$ invalidate their copies of the block upon observing the invalidate transaction on the cluster bus. CMC_1 changes the block state to Invalid Remotely upon observing the GBIN on the global bus.

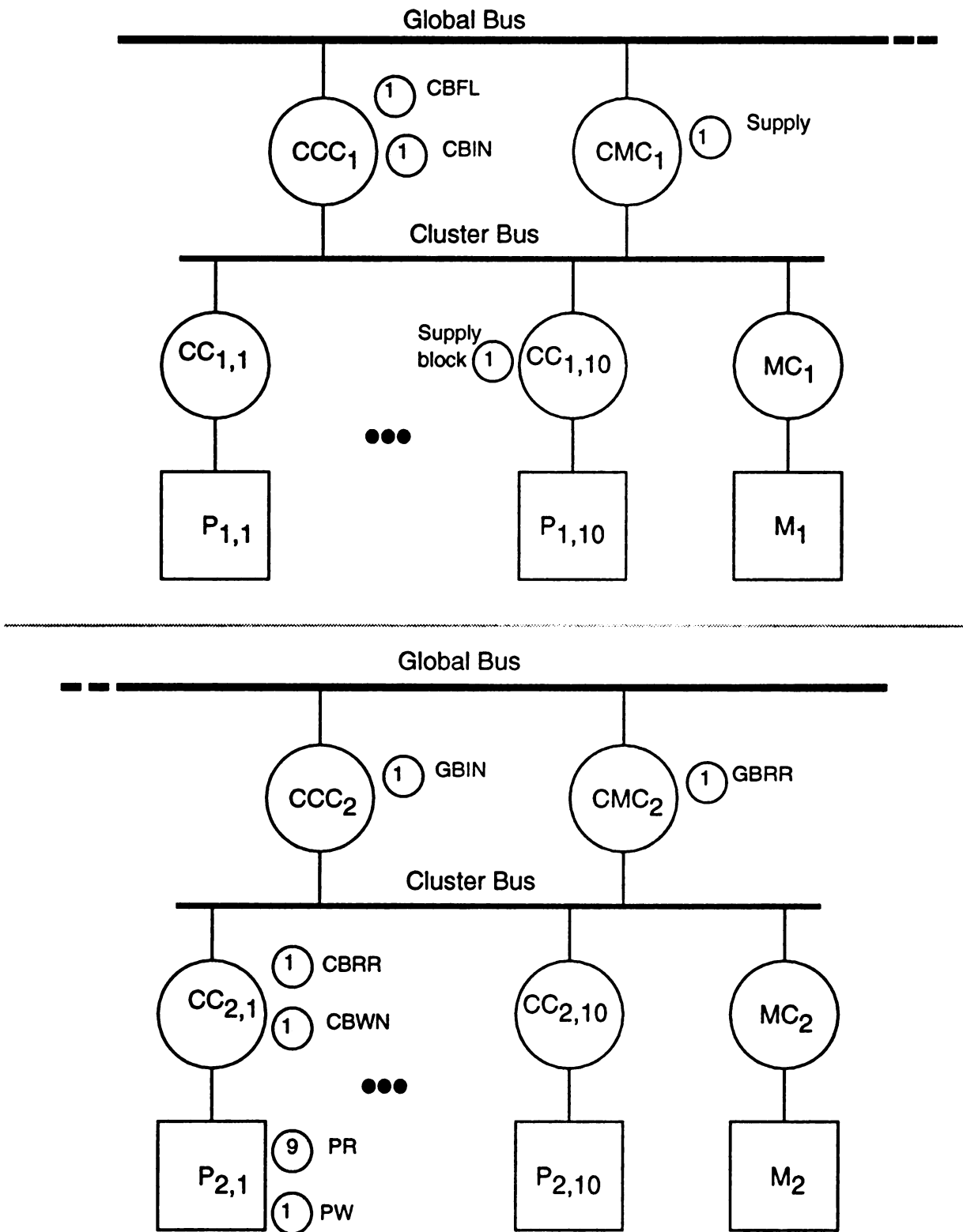


Figure 4.2: Continuation of the COGI protocol example transaction. Processor $P_{2,1}$ reads then writes the same block that is shared with Cluster 1.

4.4 Other Protocols

Archibald describes a cache coherence protocol for a two-level hierarchical bus multiprocessor in [3]. The architecture assumed by Archibald is different from the HBSM architecture by not distributing memory amongst the individual cluster buses. Centralizing memory on the single global bus seems destined to make the global bus a bottleneck. The cache coherence protocol for the cluster buses requires six states in the processor cache controllers. Nearly doubling the number of meaningful states (vis-à-vis the COGI protocol) allows Archibald's protocol to distinguish blocks held read-only, and blocks held only on one cluster. While this information can be put to good use for minimizing the effect on performance of coherence traffic, the complexity of the controllers which must implement the protocol would most likely be daunting.

Archibald's global bus protocol is very similar to COGI. The number of states is the same, and the information conveyed is equivalent. The controller for the second level caches in Archibald's system is less complex than the CCC/CMC of the HBSM since memory is concentrated on the global bus.

As part of Encore's Ultramax project [8], Wilson has developed a cache coherence protocol for a two-level hierarchical bus multiprocessor [56]. This protocol is an extension of Goodman's Write Once protocol, suitably modified for a hierarchical bus architecture. A limitation of this approach is that Write Once does not support dirty sharing of data. Thus heavy intracuster read/write sharing of data blocks requires that caches alternately acquire and invalidate a single copy of a shared block. As shown in [2], Write Once is inferior to protocols which support efficient dirty sharing, as does COGI, particularly when the level of sharing is high.

Chapter 5

Multiprocessor Address Traces

Trace-driven simulation is an important technique for performance evaluation of multiprocessor computer systems. Traces are difficult to obtain for existing multiprocessors because of their relative scarcity and the complexity and cost of the address tracing process. In order to evaluate the performance of the HBSM multiprocessor, and as an aid to debugging the COGI cache coherence protocol, a technique for generating architecture-independent multiprocessor data address traces on a widely available RISC uniprocessor was developed [5].

5.1 Introduction

Computer architects use modeling and simulation techniques to evaluate the performance of competing design solutions before physically constructing new machines. Adequate performance evaluation of the HBSM multiprocessor memory hierarchy required a trace-driven simulation approach, since the performance of the cache and memory controllers, the cache coherence protocol, and the processor-memory interconnection buses are highly dependent on the memory access patterns of parallel applications. In addition to acting as the input to a multiprocessor simulator, address traces are also useful for characterizing the memory access patterns of parallel applications for use in analytical models, such as queuing networks. Other issues which can be studied with multiprocessor traces include interconnection networks, memory paging strategies, processor instruction mixes and branching characteristics.

Multiprocessor address traces are obviously not available in the design phase of new architectures; instead, address traces from existing multiprocessors must be used. Unfortunately, since access to multiprocessors is still relatively limited and existing tracing techniques consume a large amount of the multiprocessor resource, address traces from real applications on existing multiprocessors are difficult to obtain. Another potential problem is that address traces from existing applications may not be independent of the architectural features of the machine used to generate the traces.

For example, accesses to synchronization variables will depend on the relative timing between processes, and the speed and design of memory subsystem components. If such accesses are recorded in address traces, then using those traces to study new architectures is problematic.

To avoid problems with existing traces, and to assure access to architecture-independent traces of arbitrary computer size, a software technique for generating data address traces of MIMD shared memory multiprocessors by tracing uniprocessor versions of parallel applications which utilize static task allocation and barrier synchronization was developed. This tracing technique, hereafter referred to as the Tracer, is relatively fast, portable, and does not require access to a multiprocessor. The primary advantage of this technique is that by generating architecture-independent address traces, competing architectures can be evaluated with one set of traces.

The Tracer uses the technique of source code and assembly language modification of a uniprocessor version of a parallel application to generate multiprocessor traces during execution. Reads, writes and barrier synchronization events are captured for each process of the parallel application. The Tracer has been developed on Sun Microsystems's line of SPARC computers. While the Tracer could be implemented on any uniprocessor, the relatively simple instruction set of RISC machines simplifies the process of automatically modifying the assembly language code to gather traces.

Architecture-independent address traces are the natural result of separating the tasks of trace generation and multiprocessor simulation. This dichotomy is necessary when the performance evaluation of a multiprocessor memory system is being undertaken. With such traces, the information captured for each process can not include the fine-grained temporal relationship between the traces of each process since this relationship depends on the architecture and environment present when the traces were generated. Since the programming model assumed is one of static task allocation and barrier synchronization, the multiprocessor simulator into which the traces are fed creates the actual ordering of events. Each process follows its own independent series of reads and writes until reaching a barrier synchronization event. At that time the multiprocessor simulator forces the process to wait for the other processes to reach the barrier before allowing it to continue. It is the responsibility of the multiprocessor simulator to generate accesses for synchronization variables. Within the trace of a single process, the timing of the events is constructed from the count of the number

of instructions executed since the last data access. This information is recorded in the traces. Placing high level synchronization events in the trace and letting the multiprocessor simulator construct the accesses to synchronization variables and the temporal relationship between processes allows a single set of traces to be used to evaluate competing memory hierarchy designs.

5.1.1 Tracing techniques

Several methods of gathering multiprocessor traces have been described in the literature [52]. Dubois, et al outline the general methodology of trace generation as separate and independent of multiprocessor simulation [19]. The reported techniques fall into the broad categories of hardware, software, and simulation methods.

Hardware monitors may be used to directly observe and record addresses of memory references sent to off chip caches or to main memory. Disadvantages of hardware methods include the expense of the hardware monitors, and the lack of ability to monitor memory references satisfied by internal caches. As on-chip caches grow larger, this becomes more and more of a disadvantage. In a related approach, modification of microcode may be used to record all memory references generated. By modifying the microcode on a VAX 8350, the ATUM [47] method can trace parallel applications involving up to 4 processors with only a slow-down by a factor of 20. Operating system and multi-tasking references are captured in the traces. The small number of processors and the fact that ATUM works only for the VAX architecture are limitations to this approach.

For microprocessors which support the ability to interrupt the system after every instruction, a software module interrupt handler can be used to record the memory addresses generated in each instruction. This technique is used in [21] on a bus-based shared memory multiprocessor. The main disadvantage of tracing through interrupts is the slow-down of the parallel processor being traced; each CPU may be slowed by as much as a factor of 1000.

PSIMUL is an instruction level MIMD multiprocessor emulator that has been used by IBM researchers to study memory reference behavior of parallel programs [48, 34, 6]. Simulating a multiprocessor to generate address traces is a very time consuming process; So reports a slow-down of a factor of 450 when generating traces. PSIMUL cannot trace privileged instructions, thus it may only partially capture operating

system references.

Stunkel and Fuchs describe a tracing technique for multicomputers called TRAPEDS [51]. Implemented on an Intel iPSC/2, the TRAPEDS technique is similar to the Tracer in that it modifies the application code at the assembly language level and then collects traces during execution of the program. Weinberger originally proposed the method of modifying application code at the assembly language level to record instruction counts at execution time [55]. The fundamental difference between TRAPEDS and Tracer is the focus of architectures (multicomputer versus multiprocessor) and the fact that TRAPEDS must be run on the multicomputer itself. A slow-down by a factor of 50 is reported for TRAPEDS when traces are not collected on disk; recording traces further increases the slow-down because of the I/O requirements and the hypercube architecture of the iPSC/2.

Tango is a multiprocessor simulation environment that can be used to generate data address traces for a wide range of applications and architectures [17]. The philosophy of Tango is one of providing only as much accuracy as a user is willing to pay for in simulation time. To reduce the time required to simulate a multiprocessor, Tango executes compiled sections of application code whenever possible, rather than using the more common approach of single-stepping the simulated multiprocessor through an application. This method is claimed to be 100 to 1000 times faster than more traditional multiprocessor simulators.

5.2 The Traced Application

The model assumed for the multiprocessor architecture is a very general one; multiple processors and memory modules are connected through a general interconnection network. The memory modules comprise a single shared memory address space as shown in Figure 5.1. Each process uses a portion of this shared address space as its *private* memory. Shared data can be placed in either the *global shared* portion of the address space or in the *regular shared* portion of the address space. The distinction between the private, regular shared, and global shared portions of the address space is made to facilitate the use of the generated traces in simulations of non-uniform memory access multiprocessor architectures. The meaning of the shared memory address space can be easily altered by modifying the memory mapping filter program.

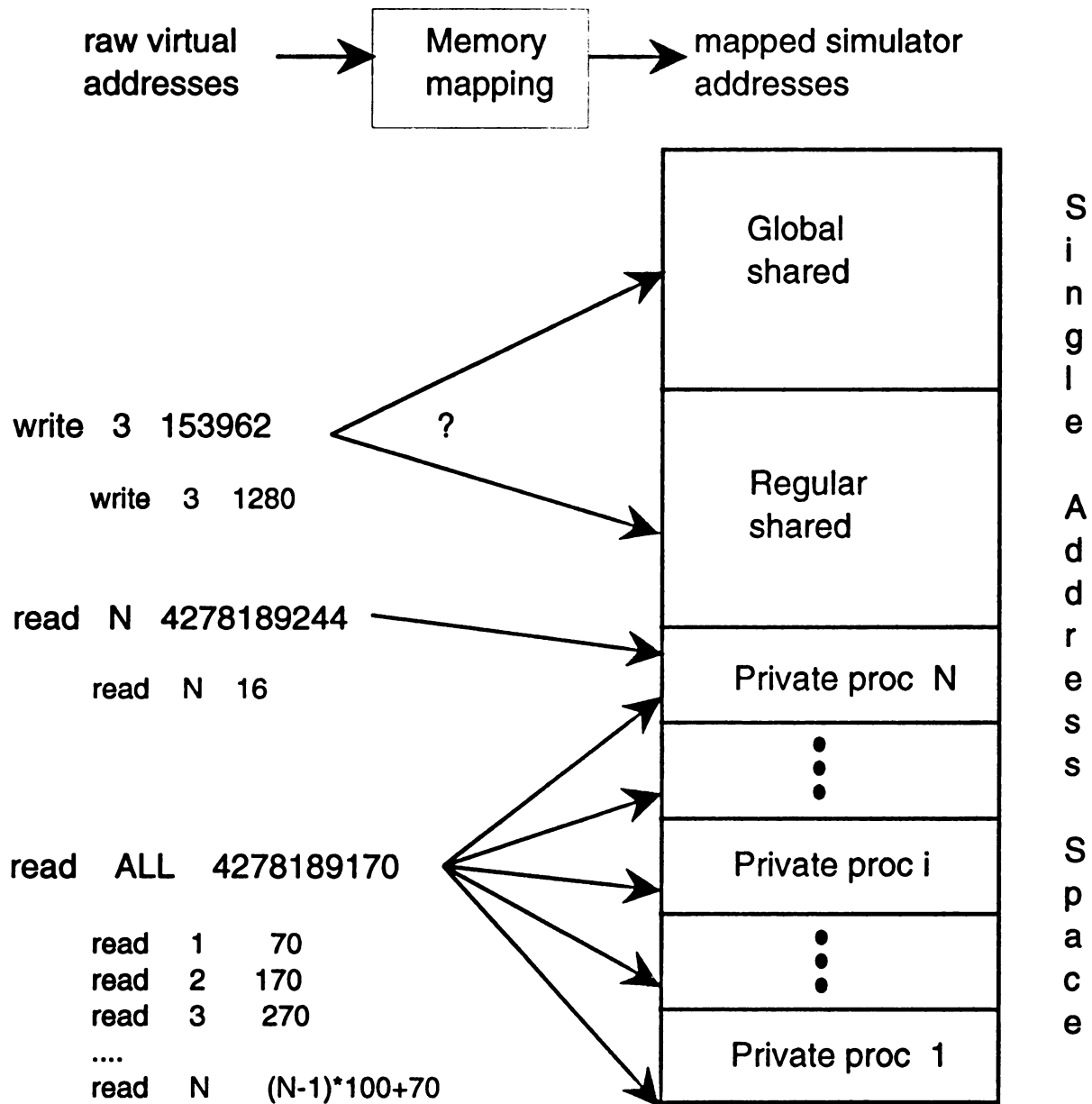


Figure 5.1: Traced multiprocessor address space. Raw virtual references are shown with resulting filtered references below them and indented.

The first step of the tracing process is to code the application in an extension of a high-level language. While the Tracer will work with any language for which process creation facilities and an adequate compiler are provided, the following discussion assumes the use of the C language. The application programs are written using the *fork* and *wait* primitives found in the UNIX operating system. The application to be traced is written in C following templates that aid the job of process creation and synchronization. The extensions to C allow each variable to be classified as either private, regular shared, or global shared. The declarations of these variables in the extended syntax are replaced by legal declarations and C function calls to record the type and virtual address of all variables at execution time.

Tracing is controlled by setting a global trace attribution variable to the identity of the process to which traced events should be attributed. The trace attribution variable can also be set to a *virtual process*. Virtual process **IGNORE** turns off tracing, virtual process **ALL** attributes a traced event to all processes, and virtual process **CONTROL** attributes the traces to a control process. The tasks of setting the trace attribution variable and defining barrier synchronizations are done with the Tracer library functions shown in Table 5.1.

attr_trace(proc)	Set the trace attribution variable to proc
def_barrier(x,y)	Defines a barrier synchronization for processes numbered x through y , inclusive
barrier(proc)	proc reaches barrier

Table 5.1: Tracer library functions for barrier synchronization and tracing.

The tasks of process creation and barrier synchronization are aided by using the code template shown in Figure 5.2. The first step in the code template is to turn tracing off with **attr_trace(IGNORE)**. A barrier is then defined for all regular processes and for a control process. The control process is used to initialize global data structures and coordinate the work of the regular processes. The for-loop with the function **barrier(proc)** in the body causes each of the regular processes to reach the barrier and wait. Tracing is then attributed to the **CONTROL** process as data structures are initialized for the regular processes. After the data structures are ready, the **CONTROL** process reaches the barrier, which releases the regular processes. The next for-loop in the template forks a child process for every regular process, 1, ...,

MAXPROC. The process identity of each child process is given by the loop variable **proc**. Each child process will generate traces which will be attributed to the process that they represent, hence the first thing a child process does is to set the tracing processor to **proc**.

A parallel application to be traced can be written by following the sequential child process creation code structure of Figure 5.2. An alternative structure is to have the control process create all children and let them run in arbitrary order. Since the traced program runs on a uniprocessor, both of these methods will generate equivalent trace files. The sequential method simplifies multiple process file I/O.

5.3 Trace Generation

Throughout this section, the traced program will be referred to by the name **app.***, using the suffixes **par**, **c**, and **s** to refer to the traced application program at different stages. The first step of the tracing process is to translate the extended syntax C language version of the application, **app.par**, into legal C source code, **app.c**. This translator is written in the AWK programming language. The Tracer next compiles the C source code of **app.c** into assembly language, **app.s**. Each of the steps in the process of tracing an application is shown in Figure 5.3. The output of each step is identified by a filename to the left of the transitions between boxes.

5.3.1 Expanding assembly language

The assembly language code in **app.s** is automatically modified to create **exp.s**. The UNIX development tools **lex** and **yacc** were used to create a program which reads SPARC assembly language and expands it to gather address traces at execution time. The added code must not interfere with the original function of the program, so extensive saving/restoring of registers is done for every point at which the original program is modified. The RISC nature of the SPARC microprocessor limits the complexity of SPARC assembly language and simplifies the creation of the expander program. The main job of the expander is to add code to **app.s** at every occurrence of load and store instructions which records the following information at execution time:

trace attribution process number

```

int proc;

set_tracing_proc(IGNORE);

def_barrier(CONTROL, MAXPROC);

for(proc=1; proc<=MAXPROC; proc++){
    wait_at_barrier(proc);

set_tracing_proc(CONTROL);

init_data_strucs();

wait_at_barrier(CONTROL);

set_tracing_proc(ALL);

for(proc=1; proc<=MAXPROC; proc++){
    if( fork() == 0 ){
        set_tracing_proc(proc);
        /* do the child's work */
        exit(0);
    }
    set_tracing_proc(IGNORE);
    if( (pid = wait(0)) == -1 )
        perror("wait");
    else
        printf("child has finished\n");
    set_tracing_proc(ALL);
}

set_tracing_proc(IGNORE);

```

Figure 5.2: C template for process creation and synchronization.

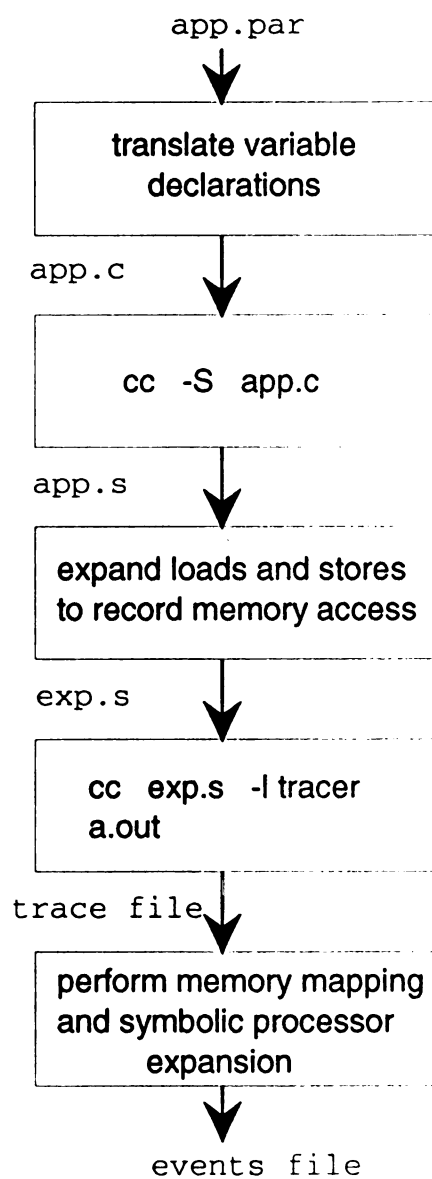


Figure 5.3: The steps of the Tracer technique.

virtual address
 value read-from or written-to memory at virtual address
 delay since last memory access

The code expansion at this step is roughly a factor of 10. Figure 5.4 shows five original lines of assembly code (at the top of the figure) which include a load instruction (`ld`) and several other instructions which do require memory accesses. The assembly language code below the dashed line is the result of expanding the original five lines of code. The original lines are marked in the expanded version with the symbols `!!!!!!`.

The value for the delay since the last memory access is kept in a global array indexed by the trace attribution processor number. Since instructions which do not access memory should execute in a predictable amount of time (ignoring interrupts and the effects of multitasking) it is reasonable to assign them a fixed “time delay” that represents the time spent by a processor in executing each instruction. For each processor, the corresponding element of the time delay array represents the sum of the delays for each of the instructions which have executed since the previous memory access. For each assembly language instruction, the expander program adds assembly language code to `app.s` for incrementing the delay value.

5.3.2 Compile and execute

After the trace gathering code is added to `app.s`, the expanded assembly code is compiled with the Tracer library routines to produce an executable program. This program is then executed to produce the raw trace file. The Tracer library routines themselves are not traced during execution. In addition, external system libraries (such as mathematical routines) are not traced. The only code that produces traces during execution is code written by the programmer and passed through the assembly language expander.

5.3.3 Filter raw traces

The final step of the Tracer takes the raw trace file as input and produces the final multiprocessor events file as output. This events file is so named because it consists of read, write, and synchronization events for each processor in the multiprocessor

<pre> L205: ld [%fp+-0x2c],%l6 cmp %l6,0x13 bge L204 nop .stabn 0104,0,269,LL100 </pre>	<pre> L205: sub %fp,0x2c,%g6 ld [%fp+-0x2c],%l6 !!!!! mov %l6,%g7 mov %o0,%g5 sethi %hi(`saveo1),%o0 st %o1,[%o0+%lo(`saveo1)] sethi %hi(`saveo7),%o0 st %o7,[%o0+%lo(`saveo7)] mov %g5,%o0 mov %l6,%o1 mov %g6,%o0 call `genrtok,2 nop sethi %hi(`processor),%g6 ld [%g6+%lo(`processor)],%g6 sll %g6,0x2,%g6 set `delay,%o1 add %g6,%o1,%g6 mov 0x4,%o1 st %o1,[%g6] sethi %hi(`saveo1),%o0 ld [%o0+%lo(`saveo1)],%o1 sethi %hi(`saveo7),%o0 ld [%o0+%lo(`saveo7)],%o7 mov %g5,%o0 mov %g7,%l6 sethi %hi(`processor),%g6 </pre>	<pre> ld [%g6+%lo(`processor)],%g6 sll %g6,0x2,%g6 set `delay,%g7 add %g6,%g7,%g6 ld [%g6],%g7 add %g7,0x4,%g7 st %g7,[%g6] cmp %l6,0x13 !!!!! sethi %hi(`processor),%g6 ld [%g6+%lo(`processor)],%g6 sll %g6,0x2,%g6 set `delay,%g7 add %g6,%g7,%g6 ld [%g6],%g7 add %g7,0x4,%g7 st %g7,[%g6] bge L204 !!!!! nop !!!!! sethi %hi(`processor),%g6 ld [%g6+%lo(`processor)],%g6 sll %g6,0x2,%g6 set `delay,%g7 add %g6,%g7,%g6 ld [%g6],%g7 add %g7,0x4,%g7 st %g7,[%g6] .stabn 0104,0,269,LL100 </pre>
------------------------------------------------------------------------------------------------------------------------	---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------	------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------

Figure 5.4: Expanded assembly code (in two columns) shown below original assembly code fragment.

to be simulated. The filter program maps virtual addresses to physical addresses, replicates process private variables, and maps virtual process number to physical process number. Virtual process number is the process to which a trace has been attributed at run time. Physical process number is the identity of the process in the multiprocessor simulation in which the traces will be used. Virtual processes are discussed in more detail below.

Addresses that are traced at run time come from the virtual address space of the uniprocessor used to generate the traces. On SPARC machines these can occur anywhere in the range 0 to $2^{32} - 1$. The filter maps these virtual addresses to physical addresses by removing the holes in the virtual address space and beginning the physical address space at 0. The type of a variable (process private, regular shared, or global shared) and its virtual address are known to the filter by reading the top of the trace file. This information is placed in the trace file at run time as a result of the declarations in `app.par`.

The virtual addresses which correspond to variables which are declared as process private are replicated by the filter so that each physical processor has a private copy of this virtual address. The arrangement of physical memory and the mapping of virtual addresses is shown in Figure 5.1. The first section of the address space represents N processor private memory spaces of P_R words each. A regular shared memory of S_R and global shared memory of S_G complete the physical address space. The distinction between regular shared and global shared memory is made to facilitate the simulation of non-uniform access shared memory multiprocessors. The filter program is written to allow the mapping of virtual to physical memory to be easily changed. The specific mapping that is used depends on the architecture of the multiprocessor that will be simulated with the traces. The arrows in Figure 5.1 show how each type of address in the raw trace file will be mapped by the filter. The first write by process 3 shows how a shared location maps into either the global shared or the regular shared portion of memory. The next read by process N represents an access to a process private variable. Finally, the read by virtual process ALL is mapped into N read events, one for each regular process. Since the variable is denoted private in `app.par`, the reads for each process are to variables in their own private address space. The accesses shown assume that the virtual address is mapped into the 7th physical address of a private address space of 100 words per processor.

5.4 Extension of Tracer

The main limitations of the Tracer are the lack of instruction traces, the lack of library and operating system traces, and the speciality of programs which can be traced. Extending the Tracer beyond its current limits is the subject of this section.

For simulation of multiprocessor computer architectures, address traces are easier to generate than they are to put to good use. The computational requirements of detailed, trace-driven simulation limits the number of memory references that can be simulated. Since data accesses typically have poorer cache behavior than do instruction accesses, the time dedicated to simulation should be spent on data address traces at the expense of instruction traces. Other applications of address traces require that instructions be traced. For example, studies of the relevant frequency of instructions can be used to design the instruction set architecture of new microprocessors. Knowledge of instruction memory reference patterns may also be needed when optimizing the design of separate instruction and data caches.

Much of what is necessary to capture instruction traces already exists in the Tracer. Since assembly code is added to a program for every instruction executed (not just the data accessing loads and stores), instruction traces could be gained by simply recording the value of the program counter at each instruction. The increase of trace storage requirements and the slow-down of trace generation would be considerable, however, since typically 75% of total memory accesses are instruction fetches. For applications which require statistics characterizing instruction or address traces, but which do not need the actual traces themselves, a process which would reduce the raw traces to statistics could be run in conjunction with tracing, thereby avoiding the time and space required to record the traces on disk.

The lack of library and operating system traces is not a fundamental limitation of the technique of the Tracer, but more of a practical problem. The Tracer can be used on any program for which the source code is available. Unfortunately, access to operating system and library source code is usually quite limited. If a particular library routine is judged important to the use of the generated traces, the library can be replaced by code written by the user thereby allowing traces to be gathered for it. Operating system traces are much more problematic. While application traces from a uniprocessor emulating a multiprocessor will be very similar to an actual multiprocessor composed of the same CPUs, the operating system of a uniprocessor

and a multiprocessor are likely to be quite different. The utility of uniprocessor operating system traces for multiprocessor system evaluation is questionable.

Generalizing the synchronization and task allocation paradigms supported by the Tracer would increase its utility considerably. Static task allocation and barrier synchronization allow the memory references streams of individual processes to be completely independent of each other except for the synchronization points. Barrier synchronization points are easily embedded in the traces. The difficulty of more general synchronization is that the addresses traced for each process will depend on the relative timing of processes as they run. This timing is in turn dependent on the environment of the computer used to generate the traces. If the intended use of the traces is to study changes to the multiprocessor environment (such as the effect of memory sub-system performance) then a circular problem has been uncovered as shown in Figure 5.5. The information in the bottom oval is only available during simulation of the multiprocessor, but the simulation of the multiprocessor clearly depends on the address traces.

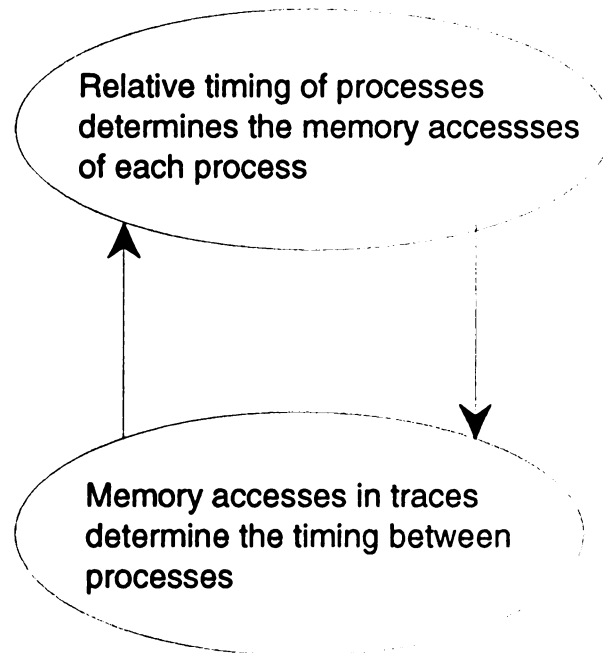


Figure 5.5: Problem of extending Tracer to general synchronization and task allocation paradigm.

One possible strategy to overcome the difficulty pictured in Figure 5.5 is to execute the Tracer and the multiprocessor simulator concurrently with communication

between the simulator and the Tracer. The feedback provided to the Tracer by the simulator could be used to determine task allocation and execution paths dependent on process synchronization. As implemented, the Tracer requires two complete passes through the raw trace file to accomplish address space mapping and virtual processor expansion. To execute concurrently with a simulator, and to receive feedback from it, the Tracer would have to be extensively modified.

Chapter 6

Petri Net Model

Detailed Petri models provide some assurance of the feasibility of the HBSM multiprocessor and COGI cache coherence protocol. Trace-driven simulations of a Petri net model served as a means of designing and debugging the COGI protocol. The Petri net models were intentionally crafted so that errors in the protocol or controllers would be readily apparent during simulation. This chapter describes the models and the High Level Timed Petri Net simulator used for trace-driven simulation of the HBSM multiprocessor.

6.1 Petri Net Modeling

A model is an abstract representation of a system which captures the important features and behavior of that system. In the design process, modeling represents a compromise between prototype experimentation and design by intuition. Constructing a working prototype of complex systems is both expensive and time-consuming. Performance results from working prototypes can be highly accurate, but the inherent inflexibility of such physical systems limits the breadth of study of possible design choices. Intuition, on the other hand, while an invaluable part of the design process, is insufficient in providing accurate estimates of the performance of complex systems. A good Petri net model provides a designer insight into the behavior of the system being modeled, supplying feedback in the design process and highlighting errors in the design.

The simple graphical structure of Petri nets allows the generation of models which are related in a clear and logical manner to their physical system counterpart. In computer modeling, relevant low level hardware details can be easily modeled with Petri nets, while at the same time less interesting functions can be ignored with more abstract net models. Petri nets are particularly well suited for modeling systems exhibiting concurrency, synchronization and competition for resources. With the increasing significance of parallel processing, the use of Petri nets for modeling and

design of computers is most natural.

Fundamental design issues common in parallel and distributed processing dictate a modeling tool which is ideally both powerful and expressive. To be powerful, a modeling tool should be based on a specification which can be related to a solid mathematical framework. The inherent power in such models is the ability to apply mathematical analysis to the model to yield estimates of system performance. Unfortunately, as Aggarwal and Gopinath [1, page 277] note,

“...as the expressiveness of the notation grows, the meaning of a specification is increasingly hard to infer in a formally precise way.”

so modeling tools which are powerful in this way tend not to be expressive enough to use with convenience on complex systems.

Simulation is an important method of extracting system performance estimates from models which are highly expressive, but either do not have a mathematical basis, or are analytically intractable. In addition, simulation is often used to understand the behavior of a system when formally determining the correctness of the system being modeled is impossible or impractical. The chief drawbacks to simulation are the complexity of developing a model and the computational requirements of running the simulation. The Petri Net simulator developed for detailed modeling of the HBSM multiprocessor addresses the first of these issues by providing a graphical and intuitive modeling paradigm. As Billington [9, page 307] notes in the context of communication protocols,

“Interactive simulation is extremely useful in the early stages of protocol specification as it allows the designer to test each part of the specification as it is completed, and can provide considerable insight into the system’s behavior. It may also be used for debugging and learning about protocols.”

Simulation of the Petri net model described in this chapter was used for designing and debugging the hardware controllers of the HBSM multiprocessor and the COGI cache coherence protocol.

The tradeoff between expressiveness (or modeling convenience) and power of the analysis techniques is aptly exhibited in the many extensions made to Petri nets. Pure Petri nets [42] can be utilized to model simple systems and study their behavior by means of techniques such as reachability analysis, method of invariants, and

transformations. For more complex systems such as communication protocols [9] or cache coherence protocols, the complexity of the system being modeled mandates the use of a more expressive modeling tool. Various proposed extensions to Petri nets facilitate the modeling of large, complex systems by increasing the expressive power of Petri nets.

6.1.1 Extensions

Pure Petri nets provide no way of distinguishing between individual tokens. In order to increase the expressive power of Petri nets, researchers [26, 32, 44] realized that a method of distinguishing between individual tokens was necessary. The two efforts in this direction, Predicate/Transition nets and Coloured Petri nets have been unified in what are known as High Level Petri nets [33].

High Level Petri nets (HLPN) attach units of information to tokens in what are called attributes. Associating attributes with tokens allows the modeling of arbitrary data as it flows through a system. Extending the transition firing rules to include conditions defined on token attribute values offers a more compact representation of powerful control structures than is feasible with pure Petri nets.

A valuable extension to Petri nets for performance modeling is the inclusion of the concept of time. Pure Petri nets, possessing no concept of time, are incapable of providing time-related performance estimates (*e.g.* throughput, utilization, availability). Work on the inclusion of the concept of time into Petri nets has proceeded in two directions: Stochastic Petri Nets and Timed Petri Nets.

In Stochastic Petri nets (SPN) [41, 57] an exponentially distributed firing delay is associated with each transition in the net. The reachability tree of the resulting net is isomorphically equivalent to a continuous time Markov chain. Well established Markov analysis techniques can be applied to yield system performance estimates. Further extensions of SPN, called Generalized SPN [38], allows the existence of both exponential and immediate transitions in Petri net models. A fundamental limitation to SPN techniques is the size of the state space of the resulting Markov chain. Because the state space grows exponentially with the size of the Petri net, SPN are limited to modeling either small systems or large systems at a higher level of abstraction. The restriction that every transition must have exponential or immediate firing delays is also a limiting factor; many systems could be more accurately modeled if constant,

non-zero delays were permissible.

In Timed Petri nets [31], a constant firing delay is associated with each transition. Geometric distributions for firing delays can be achieved by the addition of very simple subnets. Non-determinism in the state transition behavior is achieved by associating firing probabilities with each transition. With these firing probabilities, an embedded Markov chain can be defined from the Petri net and Markov analysis used to obtain performance estimates. The choice of meaningful firing probabilities and the Markov state space explosion problem are the chief difficulties in Timed Petri nets.

6.1.2 Software packages

In [13, 14] Chiola describes a software package for the analysis of Stochastic Petri nets. This package, GreatSPN, includes a graphical interface for model definition and editing. Analysis is done by constructing the reachability graph and solving for the steady state solution of the corresponding Markov chain. In addition, Monte Carlo simulation can be used to acquire estimates of the average number of tokens in each place.

Cumani [16] describes a software package, ESP, for the analysis of Stochastic Petri nets with the extension of phase-type firing distributions. The analysis strategy is similar to the one used by Chiola. To avoid the problems that are caused by introducing general firing distributions in Petri nets, the phase-type distributions allowed in ESP are introduced at the reachability tree level.

A third package for solution of SPN is described in [20]. This software package (DEEP) provides both analytical solution techniques through Markov analysis and simulation tools. In the simulation tool nets are executed for long periods of time to gain estimates of such measures as average number of tokens in a place. Dugan, et al, mention that the speed of the simulator is important, particularly for capturing rare events in a PN model.

Meta Software Corporation sells a Petri net simulation tool with features similar to the Petri net simulator described in this chapter [40]. Their Design/CPN tool includes a graphical interface, a hierarchical Petri net editor, simulation of colored Petri nets, and a \$10,000 price tag.

All of the software packages described above perform Markov analysis on the reachability tree of a Petri net. Two of them can use simulation for large or otherwise

intractable nets. However, all are limited to analysis of Stochastic Petri nets and cannot handle the extensions of High Level Petri Nets and thus are limited in their expressive power. The High Level Timed Petri Net (HLTPN) simulator used to model the HBSM multiprocessor represents a coherent and unified PN modeling tool incorporating the features of the HLPN, SPN, and TPN extensions. Petri nets of this sort are not amenable to analytical Markov solution. Instead, simulation by execution is used as an alternative for estimating time-oriented performance measures and as a design/debugging tool.

6.2 High Level Timed Petri Nets

A Petri net is a directed bipartite graph with two types of nodes: places and transitions. Connections between nodes are made by input arcs and output arcs. A High Level Timed Petri Net (HLTPN) is a Petri net with two major extensions: the concept of time is incorporated into transitions, and attributes are associated with tokens.

6.2.1 Places

Places are those nodes in a Petri net graph which hold tokens. A place can be one of two types: normal or directory. Directory places have a special structure to facilitate the efficient handling of large numbers of tokens. Normal places hold tokens in an unstructured fashion. Each place has a (possibly empty) list of attributes associated with it. Tokens residing in a place carry values for each of the place's attributes. Attributes are either single integers or arrays of floating point or integer numbers.

The marking of a Petri net at a specific time refers to the distribution of tokens among places. In a HLTPN, the state of each place can be represented with a list of the attribute values for each token present on the place. Since attributes are associated with places, each token on a place will have the same attributes, but may have differing values for them.

6.2.2 Directory places

Modeling large caches and memories in the hierarchical bus architecture requires storing a large number of individual elements to represent the state of memory blocks. Special directory places which efficiently store large numbers of tokens were developed

for this purpose. The tokens in a directory place are stored in a structured manner which provides some degree of random access to individual tokens. In contrast to normal places, where tokens are located by sequential search, a token in a directory place is accessed almost directly by the value of the ordering attribute. The cost of a sequential search of a list of possibly thousands of tokens necessitates provision of the special directory structure.

Tokens in directories are assigned to sets based on the value of the ordering attribute. Within a set tokens are stored in an arbitrary order. The process of matching a specific token in a directory (*e.g.* to check for the presence of a specific address in a cache) consists of searching sequentially through the small number of tokens in the proper set. The number of sets and the size of each set determines the size of the directory and the amount of sequential searching required to locate a token. The set associative directory can model a fully associative mapping by making the set size equal to the directory size. In this case the whole directory must be searched sequentially to locate a token. The other extreme, direct mapping, requires the set size to be equal to one; any token can be accessed directly with no sequential search.

6.2.3 Transitions

Transitions are those nodes in a Petri net graph corresponding to the occurrence of events in the model. The firing of transitions changes the marking of the net by moving tokens from input places to output places. A transition is enabled when sufficient tokens of the correct type can be assigned to each of its normal input arcs, and no tokens can be assigned to any of its inhibitor arcs. In this case “sufficient” means enough tokens to satisfy the multiplicity and conditions defined on each input arc. Three transition types are supported in the simulator: immediate, stochastic, and delayed. Immediate transitions fire as soon as they are enabled, no simulation time passes between the enabling and firing events. Delayed transitions experience a constant firing delay. Stochastic transitions experience a random firing delay selected from exponential, uniform, or Gaussian distributions.

An enabling token layer is a set of tokens on the input places of a transition which satisfy the conditions, multiplicity, and inhibition arcs of that transition. An enabling token layer allows a transition to fire. Multiple enabling token layers on stochastic or delayed transitions have unique non-zero firing delay, and hence compete with layers

of all other timed transitions in the net for firing precedence.

6.2.4 Connectivity

An input arc is a directed arc connecting a place to a transition. Input arcs of a transition are of either normal or inhibitor type. Inhibitor arcs reverse the normal sense of enabling such that the presence of proper tokens on the input arc prevents the transition from being enabled. Inhibitor arcs which act to prevent the firing of a transition are said to be activated. A transition with more than one inhibitor arc is disabled if any of its input inhibitor arcs are activated.

Enabling conditions for a transition are defined on input arcs. A condition is a predicate defined by the modeler on token attributes of input places. The predicates can be a simple comparison for equality between attribute values or a complex function defined on attribute values of input tokens. Relative conditions (between two input arcs) specify that an attribute value of a token for the first input arc together with an attribute value of a token for the second input arc must satisfy the predicate. Absolute conditions (involving only one input arc) specify that a single attribute of tokens on an input arc and a constant value satisfy the predicate.

An output arc is a directed arc connecting a transition to a place. Output arcs direct tokens to output places, and may have arbitrary, integer multiplicities. Probabilistic output arcs may be defined that send only one token to a set of output arcs.

When a transition fires, tokens from the input places are moved to the output places. For simple token mappings, the attribute values of the output place tokens are filled by copying the corresponding attribute values from a token in the enabling token layer. Complex token mappings provide a means of placing tokens on output places with attribute values which are an arbitrary function of the input token attribute values

6.3 HLTPN Simulator

The goal of the design of the High Level Timed Petri Net (HLTPN) simulator [4] was the unification of the various features and capabilities from High Level, Stochastic, and Timed Petri net extensions. The result is a highly expressive Petri net simulation

modeling tool. The simulator provides a terminal-based interface to a model editor and to a run-time environment. The simulator is written in the C programming language, and consists of 6200 lines of code in 15 files. The editor is written in the C programming language, and consists of 5200 lines of code in 15 files. Because of the numerous features included in the simulator, the syntax of a definition file is not simple. To reduce the number of syntax errors in defining a PN model, an editor was developed which guides the user through the definition process and checks on the logical format of the definition.

The objects of a Petri net model which can be created and edited are places, transitions, clean-up subnets, and directory places. Each object has a number of parameters and related information associated with it such as name, type, conditions, arcs, etc., which can be modified. In cases which a default choice is available, the editor allows the user to accept this default by hitting the carriage return. The default action at any point is indicated inside brackets just before the prompt. Whenever possible, values from the current definition of an object are provided as defaults in the editing session. This makes it easier to change small parts of a definition without re-entering all the information again. Place and transition definitions follow a line by line series of questions to which the user responds. These responses determine the course of the editor's future questions. In this way the definition effort is minimized while still allowing the user to define an arbitrary Petri net model.

The HLTPN simulator has several options which can be set at run time. These include means of controlling the display of the resulting output and the conditions under which the simulator stops. In addition, the simulator can be interrupted while running and control of the simulation regained (execution of the net stops). When the user returns to running the PN model, the simulator continues execution exactly where it was interrupted. While the simulator is stopped the marking may be examined and altered; tokens can be added, edited or deleted. The marking of the entire PN model, or of the places in a subnet, or of a single place, may be displayed.

Execution of the Petri net model consists of repeatedly firing enabled transitions and altering the net marking accordingly until execution is interrupted or stopped or until deadlock is reached. This flow of control is shown in Figure 6.1. During the first step the simulation time is not advanced. The simulator repeats the first step until no enabled immediate transitions remain unfired. The second step involves checking all timed transitions to determine those which are enabled, and firing the

one with the smallest delay value. Simulation time is advanced in this step by the delay value of the fired transition. To increase speed, the actual implementation of the simulator maintains a list of transitions which require examination in the two main loops. Initially, the list consists of all transitions. Each time a transition is examined and all enabling layers are found, it is removed from the list of transitions to check. Transitions are appended to this list as changes are made to the marking of their input places.

6.4 HBSM Multiprocessor Model

The High Level Timed Petri Net (HLTPN) model of the hierarchical bus multiprocessor is organized into 19 largely independent subnets totalling 460 places and 540 transitions. This organization eases the process of defining and maintaining the 1000 nodes of the full model with the HLTPN editor. Each subnet represents a major function of a hardware controller. For example, there are four subnets comprising the processor cache (CC): one for the processor side of the cache, one for the bus snoop portion of the cache, and one each for the cache directory and bus interface. The full model represents the complete state of each controller and of each block in cache and memory. Trace-driven simulation of the HLTPN model is slow due mostly to the level of detail of the model. The simulator can perform between 15 and 60 memory accesses per second, depending on cache hit ratio.

6.4.1 Processors

The size of the complete 1000 node model is too large to include in a publication. Instead, smaller subnets representative of the complete model are discussed. Figure 6.2 illustrates the high level actions of individual processors in the hierarchical bus multiprocessor model. In the illustrations, places which are gray are places which are depicted more than once in a figure. Each processor in the model is characterized by a cluster number and a processor number (c,p). The transition **generate** uses a complex output arc and a complex delay to generate processor memory operation events (RR-read, WN-write, SY-sync) and to model the delay between processor activity. When **generate** is enabled by a processor, the simulator uses the identity of the processor (c,p) to look-up the next trace event for that processor. Each trace event

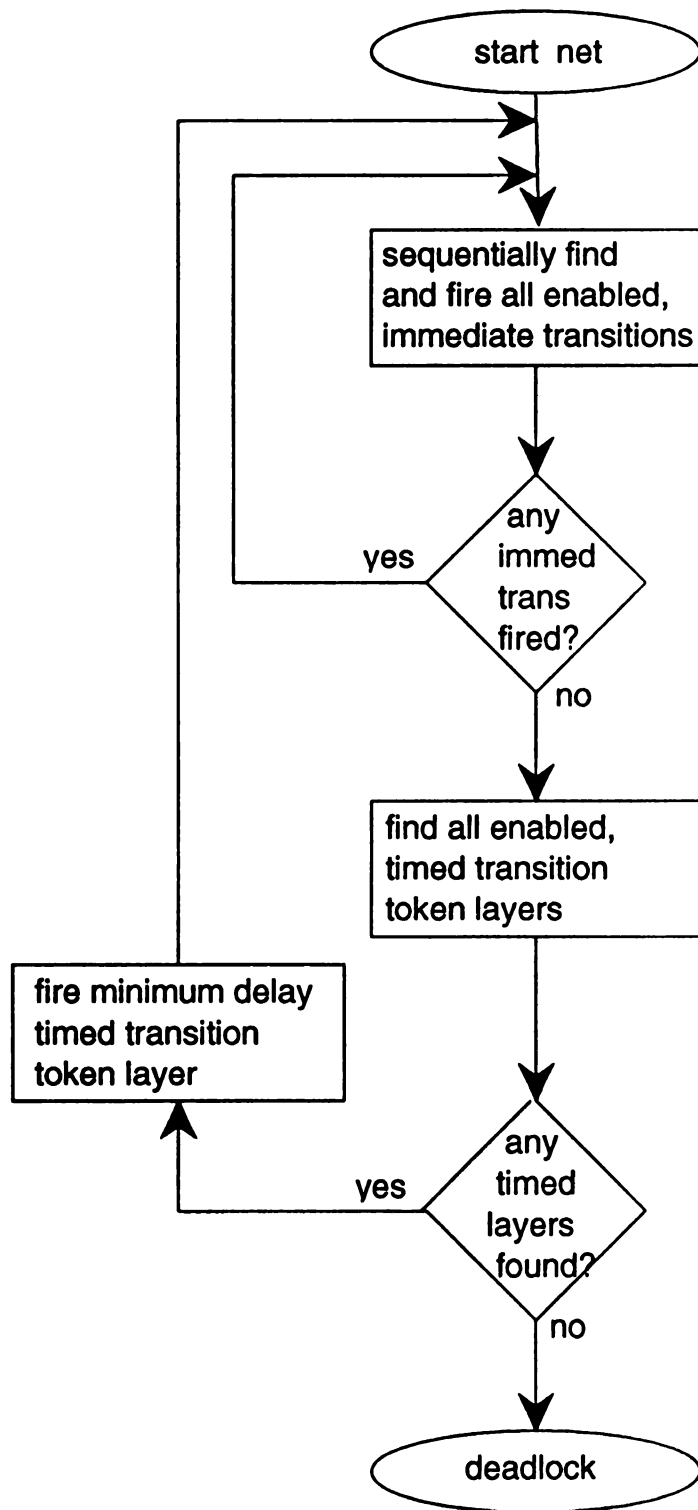


Figure 6.1: Main loop of HLTPN simulator.

includes the time that the processor spent between issuance of memory operations. This time is used as the delay value for the timed transition **generate**. The information on a token in place **requests** includes the identity of the processor (c,p), the address of the memory request (a), the type of request (r) and a value associated with this request (w). For read requests, the trace file contains the value of the memory location that was returned by the memory system at the time the traces were generated. This value can be used to test the correctness of the PN model.

Three transitions compete for tokens in place **requests**. These transitions use absolute conditions to route tokens according to the type of request. Read and write requests (RR and WN) place tokens in places **mem access in progress** and **check return value**. These transitions also send a token to the cache controller (not shown) which initiates the request in the remainder of the model (gray box). When the memory access request is complete, a token is deposited in two places in the dashed box. For read requests, transition **check return** fires on one of these tokens and compares the value returned from the model with the value expected. Transition **request done** fires for both RR and WN requests and places the processor token back in the **processors** place, thus completing a memory access event. During simulation, the simulator is instructed to stop if the transition **return bad** fires, thus preserving the state of the model near the time that an error occurred.

6.4.2 Synchronization

The subnet which handles synchronization requests is shown in Figure 6.3. This subnet performs barrier synchronization for an arbitrary grouping of processors. The attribute (y) on place **synch** represents a barrier synch number, used to identify the synchronization points. The attribute (b) is a bit field which defines the set of processors involved in this synchronization. The current status of each barrier is kept in place **barrier**. For the first processor which reaches the barrier, the place **barrier** will not have a token in it with attribute (y) which matches the token in **synch**, so the transition **middle** cannot fire, and transition **first** fires. When subsequent processors reach the barrier, the inhibition arc on transition **first** from place **barrier** prevents that transition from firing, and transition **middle** will fire instead. When either of these transitions fire, the current bit field in (b) has the bit corresponding to processor (c,p) cleared by a functional output arc. A token representing each processor is then

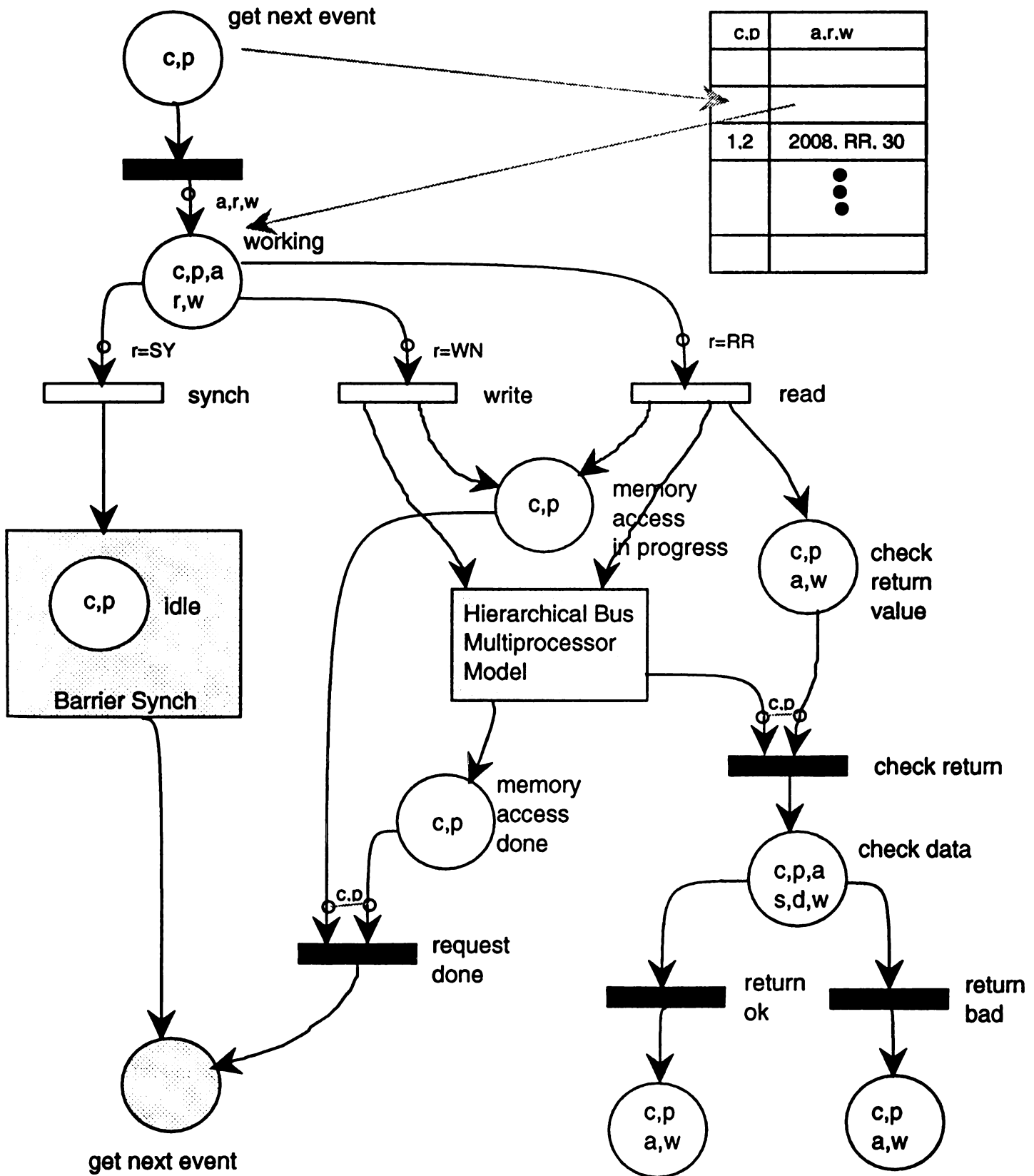


Figure 6.2: HLTPN model representing the actions of processors in the hierarchical bus multiprocessor.

deposited in place **idle** to represent a processor waiting on a barrier.

When the last processor reaches the barrier, all the bits of attribute (b) have been cleared and transition **last** fires, depositing a token into place **unlock**. Transition **unlock** uses a relative condition to match the unlocking token with each idle processor and to place them back in action again by depositing a (c,p) token in place **processors**. When all processors have been removed from place **idle**, the transition **clear unlock** can remove the unlocking token since there will be no processors on place **idle** to inhibit the **clear unlock** transition. Place **consume unlock** token is only used so that the **clear unlock** token has a place to deposit tokens. The black dot in this place represents a token without any attributes.

In the synchronization subnet the attribute (y) serves to allow multiple synchronization events to occur simultaneously in the same subnet. Being able to “fold” multiple identical subnets into a single subnet is an important attribute of High Level Petri nets for modeling complex systems. The complete HLTPN model of the memory/cache subsystem of the hierarchical bus multiprocessor folds the subnets representing each processor and cache around the attributes (c) and (p). The result is one subnet which represents a cache controller instead of $M \times N$ subnets, where M is the number of clusters and N is the number of processors on each cluster.

6.4.3 Bus arbitration

Bus arbitration is distributed amongst all potential bus masters. On a cluster bus, CCs have equal arbitration priority, and fairness is achieved among competing CCs by preventing a CC who has used the bus from participating in arbitration again until all outstanding requests have been satisfied. For this reason, bus arbitration priority can be done based on a simple CC identity. The priority of CCCs and CMCs is higher than CCs to minimize the amount of time the global bus is held by these devices. In the Petri net model bus arbitration is represented by a single subnet as shown in Figure 6.4. The current bus master selects the next bus master from those having made an arbitration request by placing a token in **select bus master elect**. If no bus arbitration requests have been made, then no tokens will be found in **bus arb requests** and the transition **at least one arb request** will not be enabled. At the same time, transition **no arb requests** is enabled, so a token is put in the place **idle bus** to indicate that the bus is idle and available. If multiple bus arbitration requests

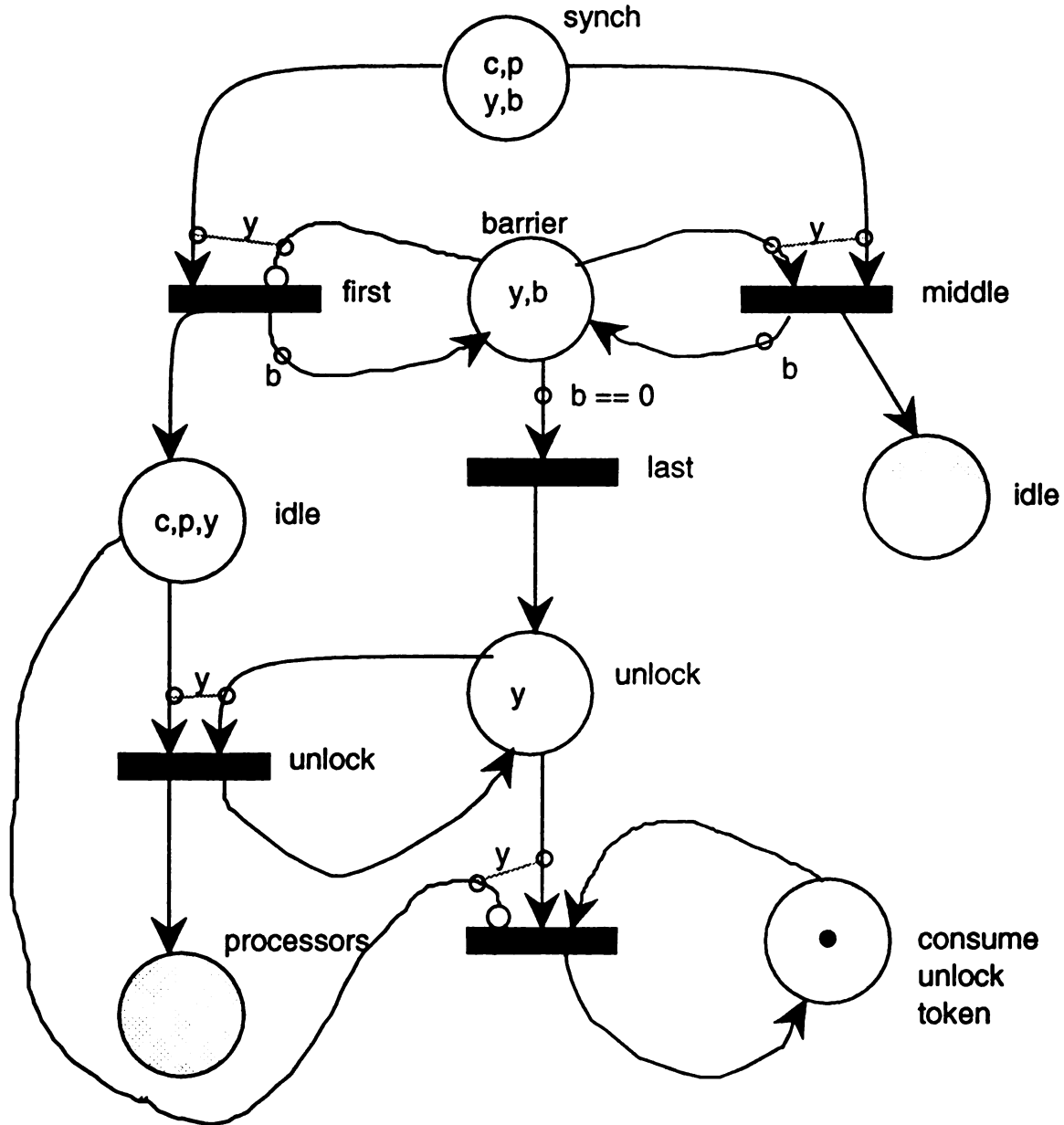


Figure 6.3: HLTPN subnet of barrier synchronization in the hierarchical bus multi-processor model.

have been made, then transition **at least one arb request** will fire, randomly picking one of the bus arbitration requests, and using the identity of that requester to initialize the attribute **m** in place **current max**. For the remaining bus arbitration requests, the transition **select max** fires and compares the attribute **m** representing the highest priority bus request to the attribute **p** representing the priority of the bus request being checked. If **p** is greater than **m**, **m** gets the value of **p** from the output arc on **select max**. Eventually, all of the bus arbitration requests in **bus arb requests** will have been checked and the maximum priority (identity) will have been found. At that time transition **election done** will be no longer disabled by its inhibitor arc, so it will fire and place the identity of the selected maximum priority request into the **bus master elect** place.

6.4.4 Directories

The model of the hierarchical bus multiprocessor required two types of directory place. The directories for caches are accessed in the set associative manner described above. The upper portion of Figure 6.5 shows a set associative directory that can be used to model a CC cache in a 2 cluster, 2 processor/cluster model. Each set in each cache holds 4 blocks (4 way set associative). Each cache block represents 4 addresses and hence 4 data words. Tokens representing blocks are stored in the directory based on the address of the block in memory. A full model of the HBSM multiprocessor will have N processors per cluster and M clusters. The data structure for this directory is organized as a three dimensional matrix which is addressed by cluster number, processor number, and block offset (frame number). Each rectangular column in the top part of Figure 6.5 represents the cache of one processor. Each column is addressed by attributes **c**, and **p**, and blocks within a column are addressed by their offsets, or frame, **f**. Cache directories store information in the following attributes: cluster id (**c**), processor id (**p**), address (**a**), frame (**f**), status (**s**), and data (**d**). Attributes **c**, **p**, and **f** are used to organize the cache into a three dimensional data structure in the simulator. The frame attribute represents the offset of a block within a directory, and is analogous to the concept of cache frame. Matching of tokens for conditions can be done either by block address or frame number. Data is stored as an array of floating point numbers, one element for each memory location.

Cluster Cache Controller (CCC) directories are also modeled with set associative

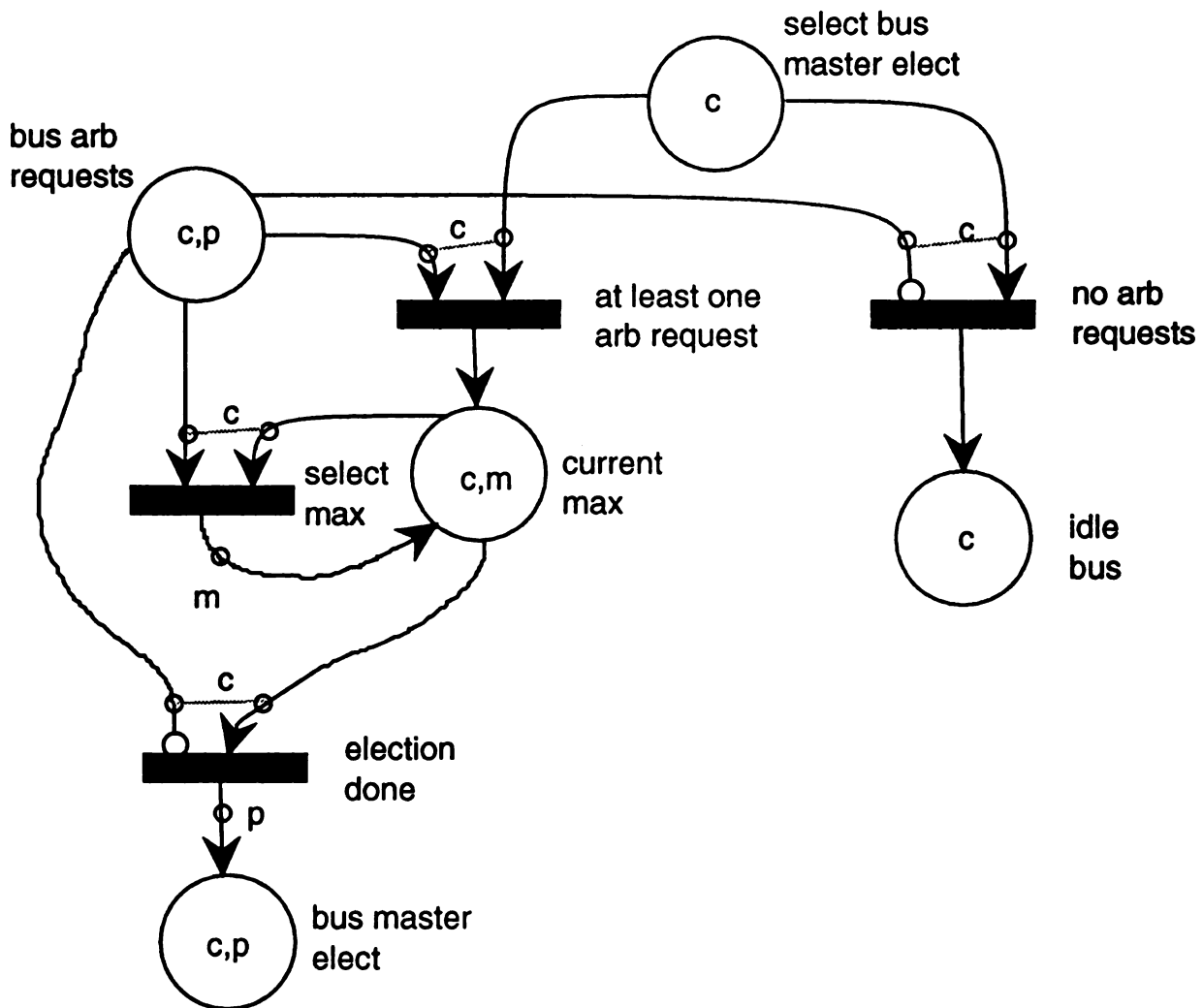


Figure 6.4: HLTPN subnet of bus arbitration for HBSM multiprocessor.

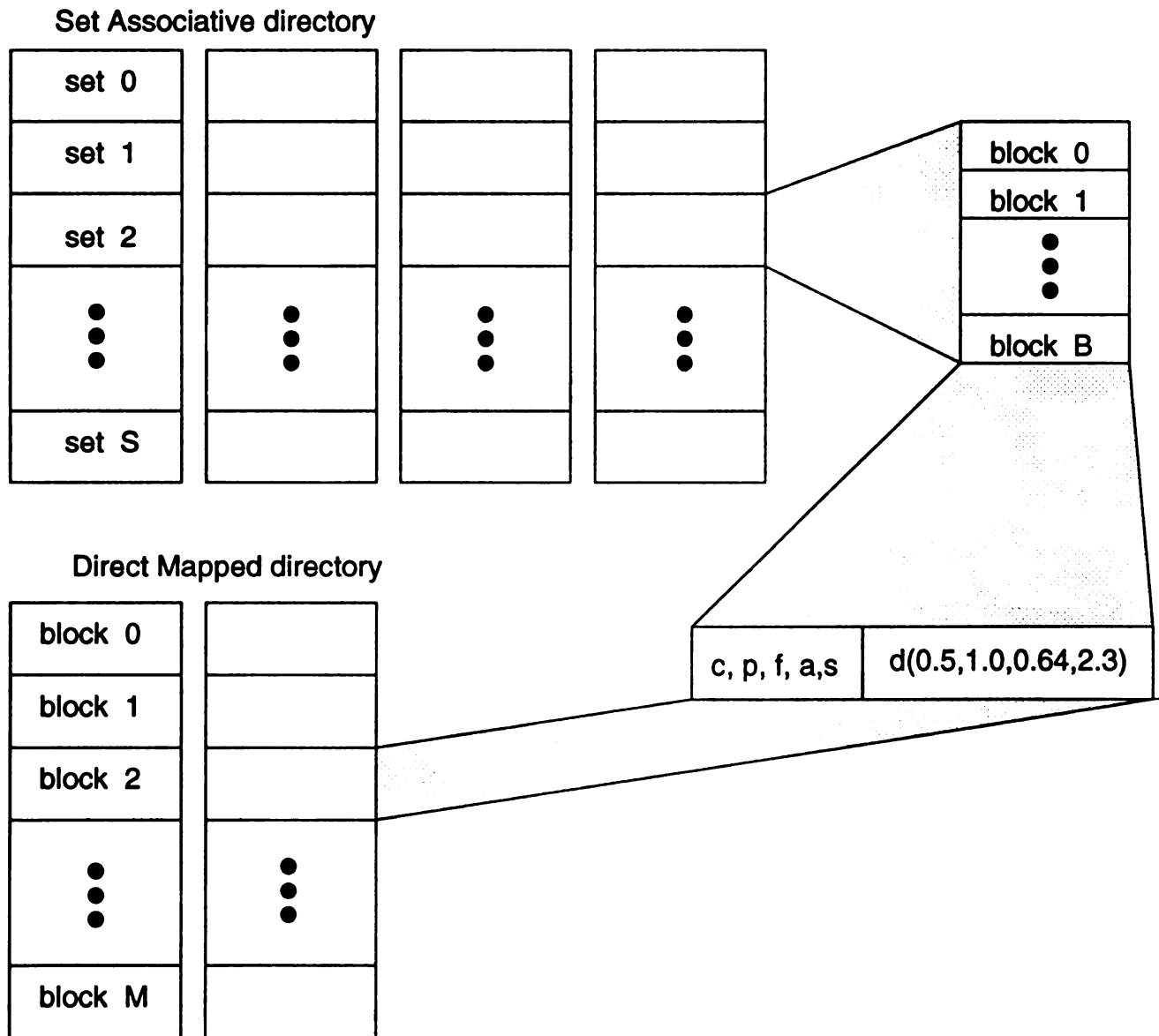


Figure 6.5: Set associative and direct mapped directories for a 2 cluster, 2 processor/cluster hierarchical cache system model.

directory places. Because there is only one CCC per cluster, there is no processor identity attribute for this directory and the indexing into the data structure is done by ordering attributes *c*, and *a* or *f*. Also, no data is held in the CCC directory.

The second directory structure is used in the Cluster Memory Controller (CMC) and the Memory (MC) to track the status of every block in memory, and to hold data for those blocks, respectively. Tokens for this type of directory represent each block of memory in the system. Each block is always represented by a token, and each directory place always has the same set of tokens, so the directory structure is organized in the direct mapped fashion (set size equal to one). The lower portion of Figure 6.5 shows this direct mapped type of directory. Access to these directories is done by block address. The Cluster Memory Controller tokens hold status information for each block while the Memory tokens hold actual data.

6.5 Design and Debugging

The initial design of the COGI protocol was checked on paper by trying to anticipate all possible scenarios and confirming that the protocol definition tables specified the correct behavior in every case. Several iterations of this method, and the consequent modifications, resulted in a protocol definition which was seemingly correct. The next step in the process was the development of the Petri net model of the HBSM memory subsystem. Modeling the active controllers revealed problems that would normally only be faced by someone planning to build a prototype machine. Some of these problems (and their solutions) are described in this section.

An asynchronous bus avoids problems of clock skew among bus masters, and is capable of operating with controllers with different response time characteristics. The handshaking that is used on asynchronous buses assures that faster devices do not outpace slower devices. This provides some measure of device and/or technology independence for the bus. In a cache coherence system which relies on snoopy controllers, each controller must be able to respond quickly enough to take actions for each bus transaction to maintain coherence. Handshaking in the form of acknowledgments for each transaction ensures that each cache controller has observed and responded to every bus transaction. If a controller is unable to respond immediately, handshaking forces the transaction on the bus to be delayed until all controllers have responded.

The handshaking used in the Petri net model is similar to that of Futurebus [49].

Two signal lines implement the necessary acknowledgments between bus masters for all transactions; DI is a many-to-one, general acknowledgment used by bus masters to signify that a transaction has been observed and responded to. DK is one-to-one signal used when one controller is sending information to another controller (*e.g.* in a cache-to-cache block transfer). Figure 6.6 shows how such bus signals can be modeled with Petri nets. The shaded portion of the figure represents the current bus master. The other transitions each represent an active snoop device on the bus. When a snoop controller sees a bus transaction it takes actions according to its responsibilities, then when it is done removes a token identifying itself from the place DI. When all tokens matching cluster attribute *c* have been removed, transition **all controllers acked, data ready** is no longer disabled by the inhibitor arc from DI. At this point, all snoop controllers on a cluster bus have responded to a transaction. When the controller which is providing data in this example has the data ready, it responds by placing a token in DK, and transition **all controllers acked, data ready** fires, finishing this transaction.

The idea of making a Petri net model which deadlocks upon error was used to help ensure correctness of the model. While running the simulator, if the model deadlocked, the state of the model was preserved close in simulation time to the point that an error had occurred. Keeping separate tokens in DI helps in debugging the protocol and model since activity on a bus stops if one of the controllers does not acknowledge a transaction, and the identity of this controller will be left in place DI. Examining the state of the model for the controller which did not respond to the bus transaction can determine the cause of the deadlock, and hence of the error.

The action of the CMC during a write to a local block is an example of a subtle bug in the COGI protocol found via simulation of the Petri net model. For a write to a local block¹ held Valid by the CMC, the CMC changes the state of the block to Invalid Locally, as shown in Table 4.5, indicating that a local CC has a dirty copy of the block. Since dirty sharing is not allowed between clusters, neither a write back nor an invalidate transaction should appear on the global bus for a block held Invalid Locally by the CMC. If the block is held Valid by the CCC, then the CCC must broadcast a global bus invalidate when it observes the cluster bus write notice transaction. However, if the cluster snoop of the CMC changes the state of the block

¹A local block is one whose home cluster is the same as the cluster on which the write notice occurs.

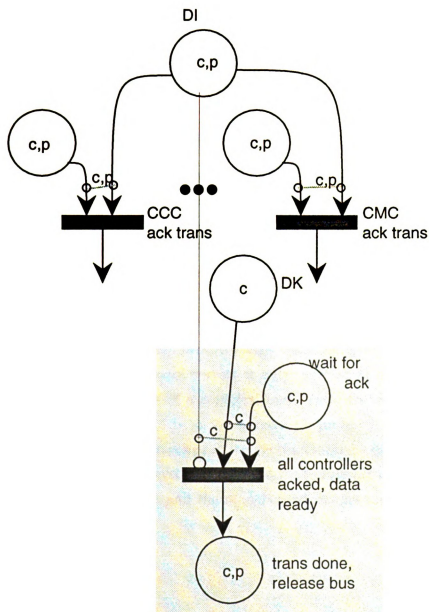


Figure 6.6: Bus handshaking signals DI and DK in Petri net model.

to Invalid Locally before the the CCC broadcasts the global bus invalidate (quite probable at higher global bus utilizations), then the CMC global snoop will observe a GBIN transaction to a Invalid Locally block, an event which seems to violate the COGI protocol. The solution, as noted in Table 4.6, is to accept such transactions, but to not take any actions upon their occurrence. Another possible solution is to have the global snoop devices (the CMC and CCC) ignore global bus transactions which originate from each other, *i.e.* the CCC and CMC of the same cluster do not snoop on each other global bus transactions.

6.5.1 Correctness

Executing the Petri net model of the HBSM multiprocessor with data address traces revealed many problems in the design of the system controllers and a few bugs in the COGI protocol. Careful design and the inherent complexity of the Petri net model combined to make the Petri net trace-driven simulations quite fragile. Errors in the design of the model and bugs in the COGI protocol resulted in a quickly deadlocked Petri net. This fragility increased confidence in the correctness of the protocol, since successful and correct execution of a large trace file seemed very unlikely in the presence of either model or protocol bugs. As discussed in Chapter 2, formally proving a cache coherence protocol correct is only possible in certain very restricted cases. Proving the correctness of a complex hardware/software system such as the COGI controllers and protocol is an open research problem. In the meantime, detailed modeling and simulation is a practical method for increasing a designer's confidence in the correctness of such systems. The specific techniques and strategies for discovering errors in the simulators and the models used in this thesis are discussed more fully in Chapter 7.

Chapter 7

Discrete Event Simulation

The second of the two part assessment strategy for the HBSM multiprocessor involves discrete event simulation for performance evaluation. While it is theoretically possible to do performance evaluation with the Petri net model, the level of detail of the Petri net model requires unacceptably large amounts of computer time. Two event scheduled simulation models were developed to estimate the performance of the HBSM multiprocessor: a probabilistic model and a trace-driven model. The faster probabilistic model is used to study the scalability of the HBSM multiprocessor. The more detailed trace-driven model is used to validate the probabilistic model and to measure the impact of coherence traffic and cache geometries on system performance.

7.1 Introduction

Scientists develop models to explain and support theories of systems. Their goal is the understanding of systems as they exist. Engineers model to evaluate changes to a system, usually with the idea of improving some aspect of a system's performance. Discrete event simulation (DES) uses a model of a dynamic system which is subject to a series of instantaneous changes of state, or events. Rapid advances in computing technology, and the ability to model complex behavior have made discrete event simulation the strategy of choice for performance evaluation of computer systems.

The two prominent DES strategies are event scheduling and process interaction [36, 23]. Both strategies involve *entities* which flow through the system (jobs, customers, bus requests, etc), *activities* which require significant time for entities, *events* at which the system changes state, and *resources* for which the entities compete.

In event scheduling, the modeler first identifies the events in the system at which the model changes state. These instantaneous events bracket system activities during which simulation time passes. For example, in an event scheduled simulation of a bank, a simple model may have only three events: entering a queue for a teller, beginning service with the teller, and leaving the bank after service. If the time taken

by a customer to select a teller was significant, then a new event could be added to the simulation to represent the customer entering the bank. These four events would then bracket the three activities during which simulation time would pass for a customer in the bank: selecting a teller, waiting for service, and service with the teller. The entities (customers) would use the services of the resource (teller) in a method described in the simulation program.

The simple structure of an event scheduled simulation program is a double-edged sword. For high level models, or for models of simple systems, an event scheduled simulation program consists of merely maintaining a future event list (in order of simulation time) and processing the events in list order. Processing events will, among other actions, add events to the future event list by scheduling them for some time in the future. The main loop of an event scheduled simulation in pseudo-code based on the C programming language would look like Figure 7.1.

If the model were to be made more complex new cases would simply be added to the single `switch()` statement corresponding to the new events. This flat structure becomes unwieldy for complicated models with many events. An additional problem with event scheduled simulations of complex systems is the temptation to lump together events which occur at the same time. For example, in Figure 7.1, no simulation time passes between the events `Select_Teller` and `Start_Service` once a customer gains a free teller¹. Without changing the behavior of the model, the `Start_Service` event could be eliminated, and the `Service_Done` event could be scheduled directly from the `Select_Teller` event. While this reduces the number of events, it makes the model more difficult to enhance and the structure more difficult to understand. Service time would no longer be clearly bracketed by start and end events, and extending the model may multiply the “entry points” of a job starting service. Without care, large event scheduled simulations can resemble unstructured computer programs in which the only control structure is the dreaded `goto`.

One of the reasons that large event scheduled simulations can become unwieldy is that the modeler is essentially ignoring what is known about the sequential relationship between the events that a customer experiences. While the modeler probably thinks in terms of the life-cycle of a customer as it flows through the system in order to decide on which events should be modeled, the flat nature of the event scheduled strategy does not allow this information to be used to structure the program.

¹This can be seen by the zero argument in the `schedule(Start_Service, 0);` function call.

```

#define Enter_Bank 0
#define Select_Teller 1
#define Start_Service 2
#define Service_Done 3

Entity    customer;
Resource  teller;
int       event;

while( (event = cause()) != NULL ){

    switch( event ){

        case Enter_Bank:
            /* decide which teller to use */
            customer.schedule(Select_Teller, select_time);
            break;

        case Select_Teller:
            if( teller.reserve(customer) )
                customer.schedule(Start_Service, 0);
            else
                /* join teller's queue */
                break;

        case Start_Service:
            /* use the teller */
            customer.schedule(Service_Done, service_time);
            break;

        case Service_Done:
            teller.release(customr);
            customer.schedule(Enter_Bank, arrival_time);
            break;

    }
}

```

Figure 7.1: Simulation loop for event scheduled DES using C-like pseudo code.

The second popular strategy for discrete event simulation is known as process interaction. The process interaction strategy uses the life-cycle of an entity flowing through the system to describe the model in the simulation program. The events experienced by an entity are embodied in a process. To continue the bank teller example, a process would be used to represent each customer in the system, and these processes would interact with each other by competing for the system resources. If the bank model needed to be extended to evaluate the results of different policies used by tellers, then the tellers could be represented by a process as well. The process interaction strategy has a more complex control structure, since multiple independent processes must communicate and coordinate their actions. Operating system support for process creation and synchronization are generally required. The advantage of the process interaction strategy is seen for complicated models which must be extended, improved or maintained. In a properly designed process interaction simulation changes to one aspect of the model are confined to the code describing the process which must be changed. The increased structure of the simulation code makes process interaction simulation easier to understand, and hence to extend and maintain. The Petri net model described in Chapter 6 is a process interaction simulation model.

In a book describing a set of tools for doing event scheduled simulation, MacDougall advises that the process interaction strategy is “strongly recommended for implementing large-scale simulation models” [36, page 4] whereas event scheduled simulation is best suited for small and medium-scale models. He cites as advantages for process interaction simulations the similitude of model and system and the hierarchical nature of the languages supporting the process interaction strategy.

7.2 Simulation Environment

In order to study large configurations of the HBSM multiprocessor, and to study those systems with large trace files, two simpler event scheduled models were developed. These models are less computationally intensive than the Petri net model by virtue of not representing the same level of detail for the controllers, buses, caches and memory modules. The event scheduled simulations are used to estimate the scalability of the HBSM multiprocessor. Both models are built on the SILO [37] libraries for event scheduled discrete event simulation using the object-oriented C++ programming

language.

The SILO libraries are C++ implementations of MacDougall's C language SMPL simulation primitives [36]. SMPL/SILO provide for *entities* which schedule *events* and *resources* which are reserved/used/released by entities. An attractive and unusual feature of this simulation environment is the implicit queueing done for busy resources. When an entity attempts to secure a resource for its use (by "reserving" it) and finds the resource is already being used by another entity, the blocked entity automatically joins a queue of entities waiting for this resource; no explicit queueing action is needed in the modeler's simulation program. When the resource is released this blocked queue is checked, and if an entity is present the event in which the blocked entity originally tried to reserve the queue is immediately re-scheduled. In other event scheduled languages the modeler must handle the joining of a blocked queue explicitly each time a resource is found to be unavailable. Because a reserve of a busy resource will cause an event to be automatically re-scheduled for some time in the future, implicit queueing requires that the blocks of code representing events must be reentrant. This requirement is readily met by forcing all event code blocks to begin with a reserve function call made inside an if/else clause, and by creating separate events for reserving multiple resources.

The method of hierarchical decomposition was followed to develop the SILO models. This technique starts with very simple models which can be easily written and debugged, and gradually adds detail until the model adequately represents the system being modeled. Each stage the model is tested and verified by simulating cases for which performance results are predictable. The development of the simulation model in stages of increasing detail harmonizes well with actual development of the simulation program. By starting with a simple and abstract model, initial programming effort can be devoted to developing a sound and logical foundation for the simulation program. The trace-driven model was developed in this way by extending the purely probabilistic model.

A major advantage of the hierarchical decomposition method is accommodation of the almost inevitable errors and dead-ends pursued in large software projects. If in the initial course of model development a fundamental flaw is recognized in the programming approach, then abandoning the flawed approach is much less wasteful (and hence more likely) when using hierarchical decomposition since the flaw has been recognized before large amounts of time have been spent on adding details to

the model. The development of the final SILO models of the HBSM multiprocessor were preceded by several of these back-to-the-drawing-board reformulations of the fundamental modeling approach.

7.2.1 Multiprocessor models

The models of the HBSM multiprocessor used for discrete event simulation are consistent with the paper design of the controllers and protocol discussed earlier in Chapters 3 and 4. Developing a detailed process interaction Petri net model of the HBSM multiprocessor required solutions to problems that would normally not arise in a simulation study. Some of these problems required solutions which had negative implications for overall performance. For example, bus utilization could be reduced by allowing a processor and cache controller to be released as soon as a cluster bus write notice transaction had been made, even if that write notice transaction required that a global bus invalidation transaction be generated. However, releasing the processor and cache controller in such a situation greatly complicated the problem of maintaining coherence in the system as a whole, so the Petri net models of the controllers do not take this course of action. The result is lower performance overall (from higher bus utilization) but a more feasible system. The event scheduled simulation models implement the same policies as those used in the Petri net model, even though the level of detail in these models would not necessarily reveal errors that may occur as a result of glossing over these types of problems. The results from the performance evaluation models will be consistent with the realistic design of the Petri net model.

The models were developed by considering all possible transactions starting from the point of view of a processor and following the COGI protocol tables. Work is done in the models by active entities representing the processors and cache controllers of the system. These entities reserve the resources of the model (global bus and memory, cluster buses and memories, caches) and perform the actions dictated by the COGI cache coherence protocol. The models are driven by memory access transactions generated probabilistically or taken from a trace file. A processor entity generates a memory access transaction then requests that its cache controller entity satisfy the memory access request. The cache controller first checks whether a memory request is a hit or miss in the cache. For a cache hit, the cache controller schedules an event at a time in the future which represents both the cache data retrieval time as well as

the time that a processor would actually spend performing operations with the word of memory. When this time has passed a new memory access event is scheduled for the processor entity.

Cache misses require that a cache controller participate in bus arbitration to gain access to the cluster bus. When the cluster bus has been successfully reserved, the cache controller checks for the case of a dirty copy of the desired block being held in another cache on the cluster. If the block is not found in another cache, then either the cluster memory is reserved and the block is retrieved, or if the block is a global memory block then the cache controller attempts to reserve the global bus. Once the required resources are reserved, future events are scheduled to release the resources held (memories and buses) after the appropriate amount of time.

Instructions and data have very different cache performance characteristics. Since instruction fetches are the majority of memory accesses of modern microprocessors, simulation time can be reduced considerably for trace-driven simulations by simplifying the model for instruction fetches. In both the probabilistic model and the trace-driven model, processors with separate instruction and data caches are assumed. Instruction caches are characterized by a hit ratio of 99%. Assuming that no writes occur to instruction caches, no write backs of dirty instruction cache blocks are needed on replacement. Misses in the instruction cache are handled just like misses in the data cache: the cache controller requests the cluster bus and reads the missing block from cluster memory. Instruction fetches are always for local blocks, *i.e.* it is assumed that an intelligent loader places the code segments for processes in the memory of the cluster on which the process executes.

The second level Cluster Cache Controllers CCC, which cache the status of all blocks found in the first level CC caches, are modeled only to the extent to which they participate in global bus activity. CCCs are assumed to have a large enough cache size and associativity that invalidations in the CC caches due to CCC replacement are negligible. Not modeling the CCC caches accurately does not have a direct impact on the performance of the system because the CCC caches are status-only, thus no benefit is gained from them during read misses.

7.2.2 System parameters

The speeds and capacities of caches, memories, buses, and microprocessors are key technology-dependent factors to the overall performance of the HBSM multiprocessor. All simulations in this chapter use the same values to characterize the performance of the hardware components. It is assumed that microprocessors issue memory references for words of 32 bits, or 4 bytes. When not otherwise specified, caches and memories operate on blocks of 4 words. Memories are assumed to be interleaved in such a way that all words of a block are available in parallel in the time it takes for memory to read/write one word. No buffering is assumed for memory, thus a read and write require the same amount of time. Processors are assumed to operate with a 50MHz (20ns) clock rate, and caches are able to supply data quickly enough to not stall the processor on a cache hit. The cluster and global buses are assumed to be 4 words wide (128 bits) and to operate at the same 20ns clock rate used by the processors. The time to transfer a 4-word block across the bus would thus be 20ns. These buses do not use a split-transaction protocol so buses are held throughout the duration of a transaction. These parameters are summarized in Table 7.1.

Parameter	Value
word size	4 bytes
cpu cycle time	20ns
bus width	4 words
bus cycle time	1 cpu cycle
cache read hit	1 cpu cycle
cache write hit	1 cpu cycles
cache directory lookup	1 cpu cycle
memory read	4 cpu cycles
memory write	4 cpu cycles

Table 7.1: Technology dependent system parameters for simulation model.

For a cache miss, the time required for a memory reference depends on the delay experienced contending for the cluster (and possibly global) bus, but for a lightly

loaded system is equal to the cache directory access time, plus the bus transaction time for a read request, plus the memory read time, plus the block transfer time across the bus. Cache misses that are writes are treated as read misses followed by write hits. For a cache hit write access the processor experiences the delay for the cache write (one cpu cycle) and the time taken to perform the necessary coherence transactions (cluster bus write notice and possibly global bus invalidate), if any.

For the purely probabilistic model, memory accesses of all types are generated randomly following distributions common to modern RISC architecture microprocessors and shown in Table 7.2. The probability of a memory access being either an instruction fetch, data read or data write is computed by dividing the frequency of each access type by the sum of the frequencies. For the hybrid trace-driven/probabilistic model the data accesses are for specific addresses and are taken from the trace file. Instruction accesses are based on the number of instructions executed since the last data reference. Addresses for instruction fetches are not used; a separate instruction cache is characterized probabilistically by a hit ratio.

Parameter	Frequency	Probability
instruction fetches per instruction	1.2	0.75
read data accesses per instruction	0.3	0.1875
write data accesses per instruction	0.1	0.0625

Table 7.2: Probabilistic representation of memory accesses.

7.2.3 Measures of performance

The number of clock cycles per instruction (CPI) is a commonly used metric which reflects both the architecture of a processor and the delays experienced due to other system components such as the memory hierarchy. When studying the impact on performance of the memory subsystem, it is useful to calculate the theoretical best possible value of CPI. Assuming that instruction and data cache accesses can not be overlapped, that multiple instructions are not initiated simultaneously, and that a cache access can be satisfied in one cpu cycle without stalling the processor, the best that can be expected for CPI is the sum of the memory access frequencies shown in Table 7.2 multiplied by the average time in cpu cycles required to execute instructions.

Assuming that each instruction theoretically takes 1.0 cpu cycles (isolated from the memory subsystem), the processors in the probabilistic model have a lower bound of 1.6 CPI. Queuing delay for buses and the need for coherence traffic can only increase this value.

The cycles per instruction metric can be used in a multiprocessor by dividing the time required for all processors to execute a certain number of instructions by the average number of instructions executed per processor. This metric will be referred to as multiprocessor CPI, or MCPI; it reflects directly the length of time required to execute a fixed length program, but does not include the effects of algorithm efficiency in the way that a speed-up measurement does. As with any average, the value of MCPI is more meaningful if all processors execute approximately the same number of instructions. Multiprocessor CPI can be used to study the degradation of system performance due to memory hierarchy performance. If the MCPI for a memory configuration is twice its best, theoretical value then the speed-up attained with an ideal parallel algorithm will be at most 50%. If MCPI is close to its lowest possible value, then the memory hierarchy will at least allow high performance and efficiency, and performance will depend on the parallel algorithm used.

When MCPI is normalized by the best possible value it can attain (the theoretical MCPI with no memory hierarchy degradation) and the reciprocal is taken, the resulting number is a measure of the potential efficiency of the multiprocessor. High potential efficiency (close to 1.0) means that the memory hierarchy does not limit the speed-up of a parallel application. Low potential efficiency (close to 0.0) means that even the best parallel application will attain no speed-up. The potential efficiency multiplied by the number of processors in the multiprocessor being simulated gives the maximum attainable speed-up.

The single most important resource in bus-based multiprocessors is the bus bandwidth. As a result, bus utilization is an important performance metric. When reported for cluster buses, utilization is an average of all cluster bus utilizations in the system being simulated. The saturation level of a bus is the utilization level beyond which it is not feasible to operate the multiprocessor. The reference saturation level shown in all plots is 75%. The MCPI and potential efficiency are used to confirm that this level of bus saturation provides reasonable system performance.

Cache hit ratio is of prime importance in reducing bus utilization and avoiding

lengthy memory access processor latency. Cache hit ratios as reported for the trace-driven model are averages across all processors in the system. All cache hit ratios refer to data caches, since instruction accesses are modeled probabilistically in both models. Write accesses which miss in the cache are treated as a read miss followed by a write hit, and thus contribute to both the hit and miss ratios.

7.2.4 Correctness

Two major questions arise when using simulation for performance evaluation. First, is the model correct, *i.e.* does it accurately represent the system being modeled? And second, are the simulator tools and the simulation program itself error free? The strategies employed for increasing the likelihood of the correctness of the simulators and models are described in this section. Most of these techniques were also applied to the Petri net simulation.

The likelihood of the simulation library being error free is enhanced by the fact that it is a widely used simulation tool. Building on the work of other researchers by using existing simulation support libraries benefits the modeler by leveraging the experience and debugging effort of many other users of the library. Confidence in the correctness of the SILO libraries and the simulation programs developed for the study of the HBSM multiprocessor is also increased by the use of modern software engineering practices exemplified by object-oriented programming.

Choosing a widely available simulation library and following good software engineering practices increases confidence in the correctness of the simulator tools to a high level. Model correctness, on the other hand, requires the application of more heuristic methods and is dependent solely on the skill and experience of the modeler. Confidence in the correctness of the model can be gained by using the method of hierarchical decomposition for model development. Building simple models that can be readily examined, understood, and tested as a foundation for later, more detailed models increases the likelihood that the models accurately represent the system being modeled. At each stage of development, several techniques have been applied to increase confidence in the correctness of the HBSM multiprocessor model.

Testing a model by simulating scenarios with predictable results is one method of confirming the correctness of a model. For the probabilistic model this involved replacing the random generation of memory access events and system interactions

with a deterministic sequence of events. For example, the function which determines whether an access is cache hit could be modified to report hits and misses in some regular pattern. For the trace-driven model, an artificially generated trace of 100 memory references by a single processor to the same block will result in an easily calculated cache hit ratio and bus utilization.

Another useful technique is that of testing the simulator at extreme values of input parameters. For example, running the simulator with a cache hit ratio of 0% and 100% or with a global access probability of 0% provides “sanity checks” on simulation results. Incorrect behavior of the model at such extremes is usually easier to detect than for more reasonable parameter values. For the trace-driven simulation model, a very effective method of finding model errors was to execute the model with very small cache and set sizes. Such extreme values of cache geometry tended to exercise the most complex parts of the HBSM multiprocessor model, such as cache replacement actions and cache-to-cache dirty sharing. These “torture” tests of small cache trials uncovered many subtle flaws in the simulation models.

To confirm the correctness of the timing of transactions, the actual simulation time for a transaction was measured and averages of these times were reported with the simulation results. Errors could be detected by finding transactions which required more or less time than their known maximum and minimum times.

During program development, a one page outline of the simulation program was maintained in parallel with the actual C++ program. This outline graphically showed the relationship between events in the model, and included the time between the events comprising each transaction. Because of its brevity and systematic layout, the outline served as a means of viewing the entire model without becoming lost in the details of the C++ implementation. The correctness of the outline of events could be more easily checked than the simulation program itself. Assuming the outline of events was an accurate representation of the HBSM multiprocessor, the correctness of the actual C++ program was reduced to the much simpler task of checking that it matched the outline of events.

The trace-driven simulation model provides more possibility for errors, since it is more detailed than the probabilistic model, but at the same time allows some important new methods for checking the correctness of the model. When the address traces are created by the multiprocessor address trace generator, the value returned by the memory system is recorded for every read access. The trace-driven simulation

model can use this information to check that the value returned to the processor during simulation is the same as the one recorded in the traces. If the returned values differ then an error has occurred in the model. This technique of uncovering errors was used quite successfully in both the Petri net simulation model and the trace-driven SILO model. A second advantage of the trace-driven model over the probabilistic model for considerations of correctness is the ability to write the simulation program in such a way that errors which occur in the model of the HBSM multiprocessor memory system are caught by the simulation program. For example, during a cluster bus write notice a cache controller must check for the presence of the block in the other caches on the cluster. If a copy is found in another cache in the private, dirty state, then a violation of the COGI protocol has occurred. Defensive programming can detect such situations so that the model error can be tracked down.

Formally proving a simulation tool or model correct is an open research problem. Appropriate strategies of design and programming, and heuristics for testing and debugging can provide a reasonable level of confidence in simulation results. Table 7.3 summarizes the techniques employed to increase confidence in the simulation results for the HBSM multiprocessor. The application of two entirely different modeling tools and paradigms (Petri nets and SILO) to the study of the HBSM multiprocessor and COGI protocol further increase the likelihood of the protocol being correct.

Technique	Applied To Model
hierarchical decomposition	both
short graphical events outline	both
extreme value testing	probabilistic
small cache torture tests	trace-driven
artificial scenario testing	both
measurement of transaction timing	both
checking returned value against trace	trace-driven
internal consistency checks	trace-driven

Table 7.3: Techniques of increasing confidence in correctness of models.

7.3 Probabilistic Model

The first model is purely probabilistic (no traces are used) and consists of multiple clusters of processors issuing three types of accesses: instruction fetches, data reads, and data writes. The stream of references for each processor is generated with a random number generator and a set of probabilities which characterize memory reference frequency in modern RISC microprocessors. Actual addresses for memory references are not generated by this model; the only distinction made is whether a memory reference is for a local cluster memory location or for a global memory location. Since actual addresses are not generated for memory references, the behavior of the cache is controlled by the hit ratio and write-back parameters supplied to the simulator. This model is a compromise between speed and accuracy. It is used to explore the range of scalability of the architecture and to form a baseline for comparisons against the more accurate trace-driven model.

An advantage of the probabilistic model is that it always operates in a steady state mode, since it is a purely probabilistic representation of the system being modeled, while results from a trace-driven simulation may be affected by the phenomenon of cache cold-starts. On an otherwise unloaded 25MHz 68040 NeXT workstation, the rate of simulation for the probabilistic model varies from 3,850 memory references per second to 850 memory references per second over the range of interest of the various simulation parameters. For the experiments done with this model each processor generated 10,000 memory references, independent of the size of the system being simulated.

The probabilistic nature of the model requires that the actual operation of the system being modeled be simplified. For example, it is impossible to model the interaction between caches for dirty sharing of a block (coherence traffic) when addresses are not actually used in memory references. Theoretically, a parameter could be supplied to the simulator to represent the probability of every sort of event occurring. Practically, it is very difficult to specify all of these probabilities since they are so dependent on the actual memory reference patterns of the processors. The strategy adopted for this first model was to specify probabilities for the most important interactions and to ignore the interactions which have only second-order effects on the overall system performance. The results of this strategy are difficult to classify as either optimistic or pessimistic; some of the simplifications made will hurt performance

and others will help performance. However, the first-order effects on performance are accurately represented, and the model can be used to quickly explore wide ranges of important characteristics and system capacities.

The parameters which are supplied to the simulator are shown in Table 7.4. The cache hit ratio parameter applies to data caches only; instruction caches have a non-varying 99% hit ratio. Data accesses are classified as being either for global memory (`global_prob`) or being for local cluster memory ($1 - \text{global_prob}$). The `dirty_replacement_prob` determines what percentage of replacements are for dirty blocks and hence generate write backs to memory. The probability that a CC cache controller must perform a cluster bus write notice for a write hit in its cache is `write_notice_prob`. For each write notice that is done, the probability that a block is also shared between clusters, and hence the CCC must perform a global bus invalidate for the block is `global_invalidate_prob`.

Parameter	Meaning
<code>hit_ratio</code>	Data cache hit ratio.
<code>global_prob</code>	Accesses for global versus local cluster memory. Also determines probability of global bus write back on replacement.
<code>dirty_replacement_prob</code>	Chance that a block chosen for replacement on a write miss will be dirty and will require writing back to memory.
<code>write_notice_prob</code>	Chance that a block is being shared by other processors in same cluster so that a write notice is needed.
<code>global_invalidate_prob</code>	Chance that a write notice must be translated into a global bus invalidate by the CCC.

Table 7.4: Probabilistic parameters for simulation of first model.

The first model was used to estimate the scalability of the HBSM multiprocessor. With five probabilistic parameters to study and two configuration parameters to control (number of clusters and number of processors per cluster), a strategy is needed to explore this large parameter state space. The strategy employed was to guess at the most important parameters, study them in isolation, pick reasonable values for them,

then experiment with full systems to verify the choice of first-order effect parameters.

7.3.1 Single cluster

It is a well known fact that in bus-based multiprocessors hit ratio is of first-order importance in determining bus utilization (and hence to overall system scalability and performance). The first experiment done with the probabilistic model was to determine the effect of hit ratio on bus utilization for a single cluster machine. No accesses to shared global memory are made in this experiment, thus no read requests, write backs or invalidates are made on the global bus. Cluster bus utilization is plotted as a function of number of processors for various hit ratio trials in Figure 7.2. The probability of dirty replacements and for dirty sharing are set at 10%.

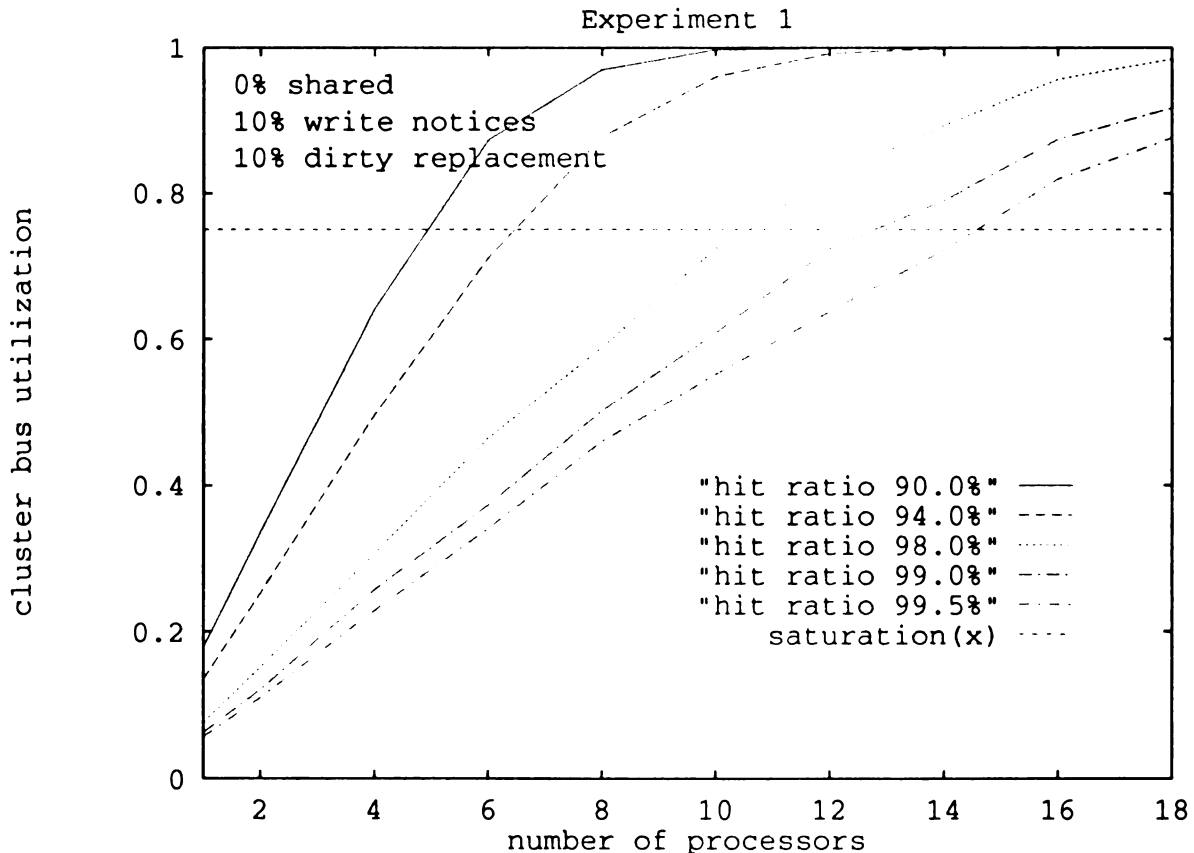


Figure 7.2: Effect of hit ratio on cluster bus utilization for single cluster multiprocessor.

The strong influence of hit ratio on bus utilization is seen by the rapid climb of the curves toward bus saturation for lower data cache hit ratios. Figure 7.2 shows

that in order to support 10 processors on a single cluster the data cache hit ratio must be 97.5%. This is a reasonable expectation for the hit ratio of a data cache for large scientific problems and will be used in further experiments to study the consequence of varying the other simulation parameters. For large, 2-way associative, LRU caches in a uniprocessor, Hennessy and Patterson [30, page 424] cite studies that show data cache hit ratios of up to 98.1%. Thus ten processors per cluster does not represent the maximum size of a cluster, since the hit ratio of 97.5% is not overly optimistic. For ideal data cache hit ratios (say 99.5%) the number of processors per cluster could be as high as 14.

As seen in Table 7.5, the average number of cycles per instruction remains in a reasonable range for cache hit ratios greater than 97.0% and number of processors per cluster less than 12. For the target system of 10 processors per cluster and data cache hit ratios of 97.5%, the MCPI is 1.90². The measure of MCPI is less sensitive to data cache hit ratios than is bus utilization because the majority of memory references (75%) are for instructions, and instructions have a separate cache with a high hit ratio (99%) and no write-backs of dirty blocks.

The first experiment established a maximum number of processors per cluster for a given cache hit ratio. To isolate the effects of global access and hit ratio, no accesses to global memory were made, and write notices and dirty replacements were kept fairly small. Three more experiments were run to determine the impact of global accesses, write notice traffic, and dirty replacements on the size of a single cluster.

The second experiment determined the effect of global accesses on the size of a single cluster. The trials shown in Figure 7.3 are for a fixed cache hit ratio of 97.5% and the same dirty sharing and replacement probabilities as in the first experiment. The close grouping of curves for a wide range of global access probabilities (1 to 25%) indicates that global accesses have only a small effect on the number of processors that a single cluster can support. The surprisingly little effect which global bus accesses have on a single cluster is due to the fact that no queuing delay is ever experienced for the global bus. Since the simulation is for one cluster, access to the global bus and memory are guaranteed once the cluster bus is acquired. Global memory accesses add only a constant time to a transaction since there is no global bus contention.

The third experiment was performed to gauge the impact of the probability of

²A perfect memory system which always supplied access to memory blocks in one cycle would have an MCPI of 1.6.

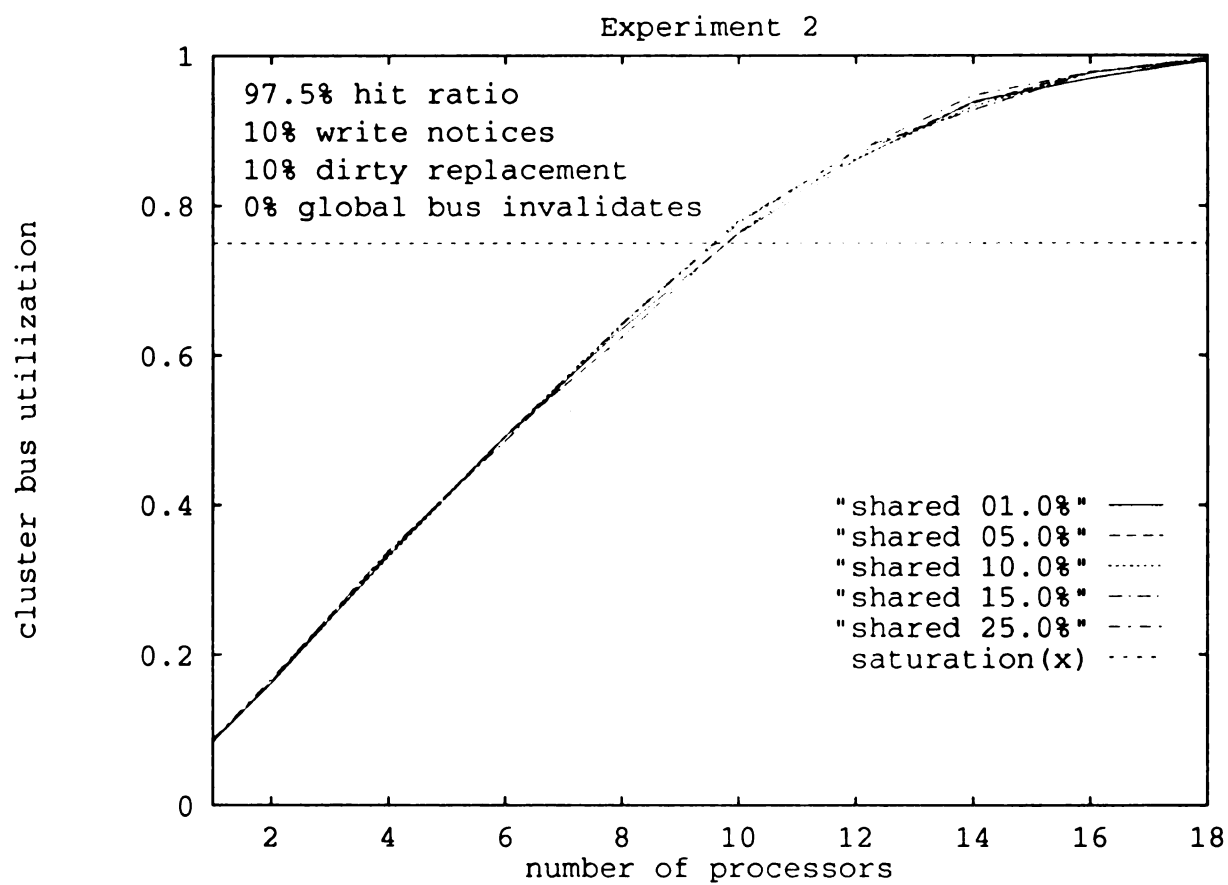


Figure 7.3: Effect of global accesses on cluster bus utilization for single cluster multiprocessor.

Number of Processors	Cache Hit Ratio								
	90%	92%	94%	96%	97%	98%	98.5%	99%	99.5%
1	2.00	1.94	1.89	1.83	1.81	1.77	1.76	1.74	1.73
2	2.01	1.95	1.90	1.83	1.81	1.78	1.76	1.74	1.73
4	2.15	2.05	1.96	1.87	1.84	1.80	1.79	1.76	1.75
6	2.40	2.23	2.05	1.91	1.89	1.83	1.80	1.78	1.76
8	2.85	2.49	2.27	2.02	1.94	1.86	1.84	1.80	1.78
10	3.48	2.96	2.52	2.16	2.04	1.91	1.87	1.82	1.79
12	4.10	3.52	2.88	2.37	2.17	1.99	1.93	1.88	1.81
14	4.89	4.08	3.32	2.68	2.35	2.08	2.00	1.90	1.84
16	5.54	4.65	3.83	2.90	2.61	2.26	2.10	1.98	1.91
18	6.19	5.30	4.24	3.32	2.85	2.41	2.30	2.09	1.96

Table 7.5: Effect of hit ratio on multiprocessor cycles per instruction (MCPI) for single cluster multiprocessor Experiment 1.

selecting a dirty block for replacement in the CC caches. The hit ratio is placed at the value selected from the first experiment, no access to global memory is made, and the level of dirty sharing on the cluster is 10%. The close grouping of curves in Figure 7.4 for a wide range of dirty write back probabilities (1 to 25%) indicates that the time spent writing back dirty replaced blocks has only a small effect on the number of processors that a single cluster can support. This effect is greater than that of the global access experiment, since the difference between the extreme curves (1 and 25%) represents almost one whole processor.

The fourth experiment was performed to gauge the impact of the probability of broadcasting a write notice on the cluster bus for each write hit to a CC cache. The hit ratio was set to the value selected from the first experiment, no access to global memory is made, and the level of dirty replacements is 10%. The fairly close grouping of curves in Figure 7.5 for a wide range of dirty sharing probabilities (1 to 25%) indicates that the time spent broadcasting write notices for intracluster shared blocks decreases the number of processors that can be supported on a single cluster by approximately one.

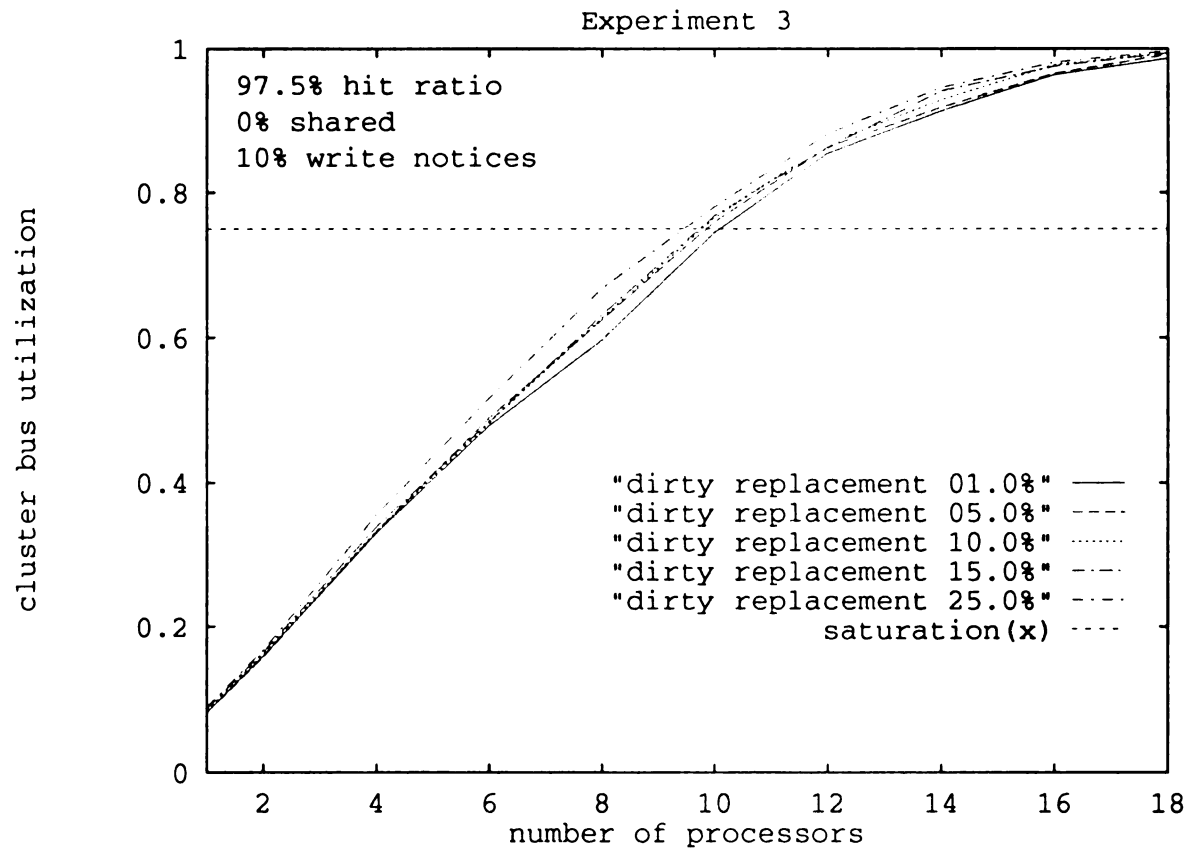


Figure 7.4: Effect of dirty write-back probability on cluster bus utilization in single cluster multiprocessor.

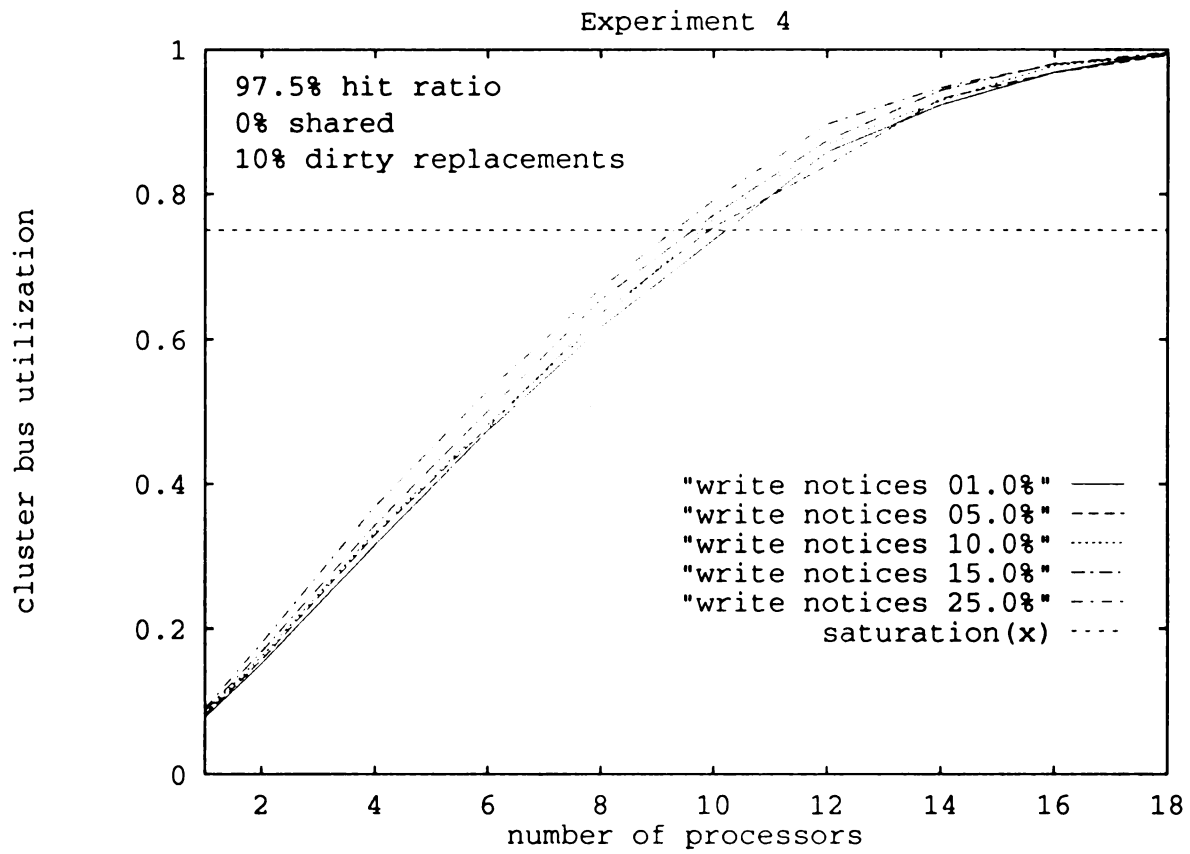


Figure 7.5: Effect of write notice probability on cluster bus utilization in single cluster multiprocessor.

Together, experiments 2, 3, and 4 indicate that some degradation in the number of processors per cluster is to be expected as a result of write notices, dirty block replacements, and global memory accesses. Since these effects will be even more pronounced in multiple cluster machines, the number of processors per cluster should be reduced to 8 in order to keep cluster bus utilization in the acceptable range.

7.3.2 Multiple cluster

The first four experiments for a single cluster identified a reasonable number of processors per cluster (8) which can be supported at a 97.5% cache hit ratio over a wide range of dirty replacement, write notice, global invalidate and global access probabilities. With the number of processors per cluster established, and a target cache hit ratio and MCPI, further experiments will investigate the ultimate scalability of the HBSM multiprocessor by simulating multiple clusters.

Ideal application

The first multiple cluster experiment uses parameters typical of the ideal type of application for a hierarchical bus architecture. This application would be typified by intraccluster read and write sharing, but by intercluster read sharing only. In other words, blocks of global memory would be shared between clusters for reading, but dirty or write-sharing of blocks would be limited to processors on the same cluster. It is also assumed that the cache geometry (size and associativity) and replacement policies are such that replacement of dirty blocks during read misses is never done.

The plot in Figure 7.6 shows the global bus utilization as a function of the total number of processors (number of clusters, C , times the number of processors per cluster, P) in the system. Since the number of processors per cluster P is fixed at 8, this is equivalent to plotting against the number of clusters, but directly shows the total size of the HBSM multiprocessor being simulated. Each line represents a trial of the experiment for a different level of shared global memory accesses. For global access levels below 10%, the global bus remains unsaturated for a 200 processor system. This is not surprising, considering the type of application that is represented by the simulation parameters used in this experiment. Highly clusterable applications should exhibit much greater intraccluster sharing than intercluster sharing, and as a result the global bus can support the needs of 26 clusters of 8 processors each.

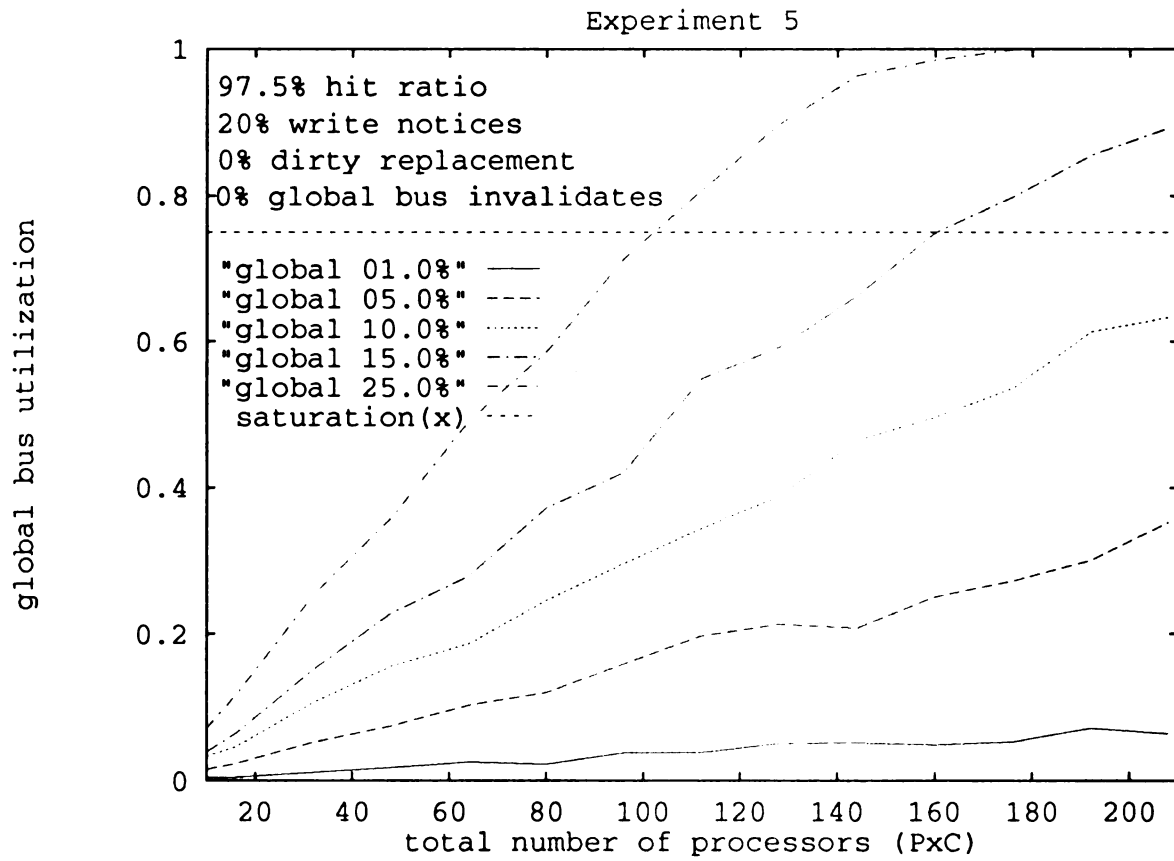


Figure 7.6: Global bus utilization in ideal application.

Though not shown in a figure, the average cluster bus utilization in this experiment remains well below saturation (around 65%) for all but the highest levels of global accesses on the largest machine. In addition, the average number of cycles per instruction (MCPI) remains almost independent of the total multiprocessor size for global access levels of 10% or less, as shown in Table 7.6. For these access levels the MCPI remains very close to the target of 1.9 established in the single cluster experiments. Figure 7.7 shows the same data when plotted as potential efficiency. For global access levels of 10% or less, the potential efficiency remains very close to its maximum value for multiprocessors of up to 200 processors. For a configuration of 208 processors (26 clusters of 8 processors per cluster), the potential efficiency of 84% means that the memory hierarchy would allow a speed-up of 175 for a perfect parallel application.

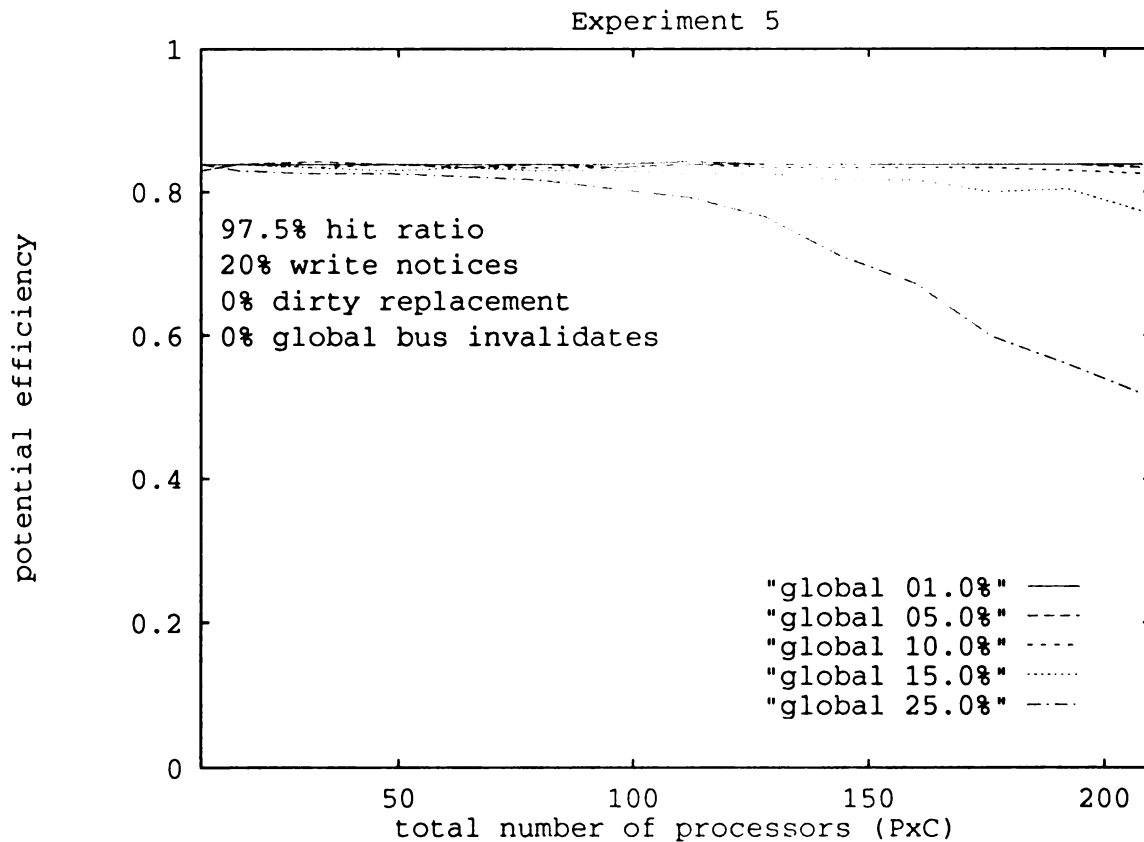


Figure 7.7: Potential efficiency for ideal application.

Total Processors (Clusters)	Global Memory Accesses							
	1%	2%	5%	7%	10%	15%	20%	25%
8 (1)	1.91	1.92	1.93	1.92	1.93	1.91	1.93	1.91
16 (2)	1.91	1.91	1.91	1.91	1.91	1.91	1.91	1.93
32 (4)	1.91	1.91	1.90	1.91	1.92	1.92	1.93	1.94
48 (6)	1.91	1.91	1.91	1.92	1.91	1.93	1.92	1.94
64 (6)	1.91	1.90	1.92	1.92	1.92	1.92	1.93	1.95
80 (10)	1.91	1.91	1.91	1.92	1.92	1.93	1.94	1.96
96 (12)	1.91	1.91	1.92	1.91	1.92	1.93	1.96	1.99
112 (14)	1.90	1.90	1.91	1.91	1.91	1.94	1.96	2.02
128 (16)	1.91	1.91	1.91	1.92	1.92	1.94	1.98	2.09
144 (18)	1.91	1.91	1.91	1.91	1.92	1.96	2.03	2.25
160 (20)	1.91	1.91	1.91	1.92	1.92	1.96	2.09	2.38
176 (22)	1.91	1.91	1.91	1.92	1.92	2.00	2.16	2.67
192 (24)	1.91	1.90	1.91	1.92	1.93	1.99	2.30	2.85
208 (26)	1.91	1.91	1.92	1.92	1.94	2.07	2.53	3.08

Table 7.6: Multiprocessor cycles per instruction (MCPI) for ideal application Experiment 5.

Non-ideal application

To gauge the effect of running applications on the multiple cluster machine which are less suited for the hierarchical bus architecture, the second multiple cluster experiment was done with a global bus invalidate probability of 25%, and less optimistic assumptions about dirty replacements of cache blocks. This application would be characterized by a fairly high level of intercluster write sharing, since one out of four write-notices would broadcast on the global bus, and cache controllers would spend more time performing write-backs of dirty local and global blocks.

As expected, global bus utilization rises more quickly and exceeds saturation at lower levels of global access than in the previous experiment. Figure 7.8 shows that with write-sharing between clusters and more block replacement traffic, the HBSM multiprocessor can only support 2% global accesses at the full size of 200 processors. Put another way, at the same 10% global access level of the previous experiment, the global bus can only sustain 16 clusters or 128 processors, a decrease in size of 38%.

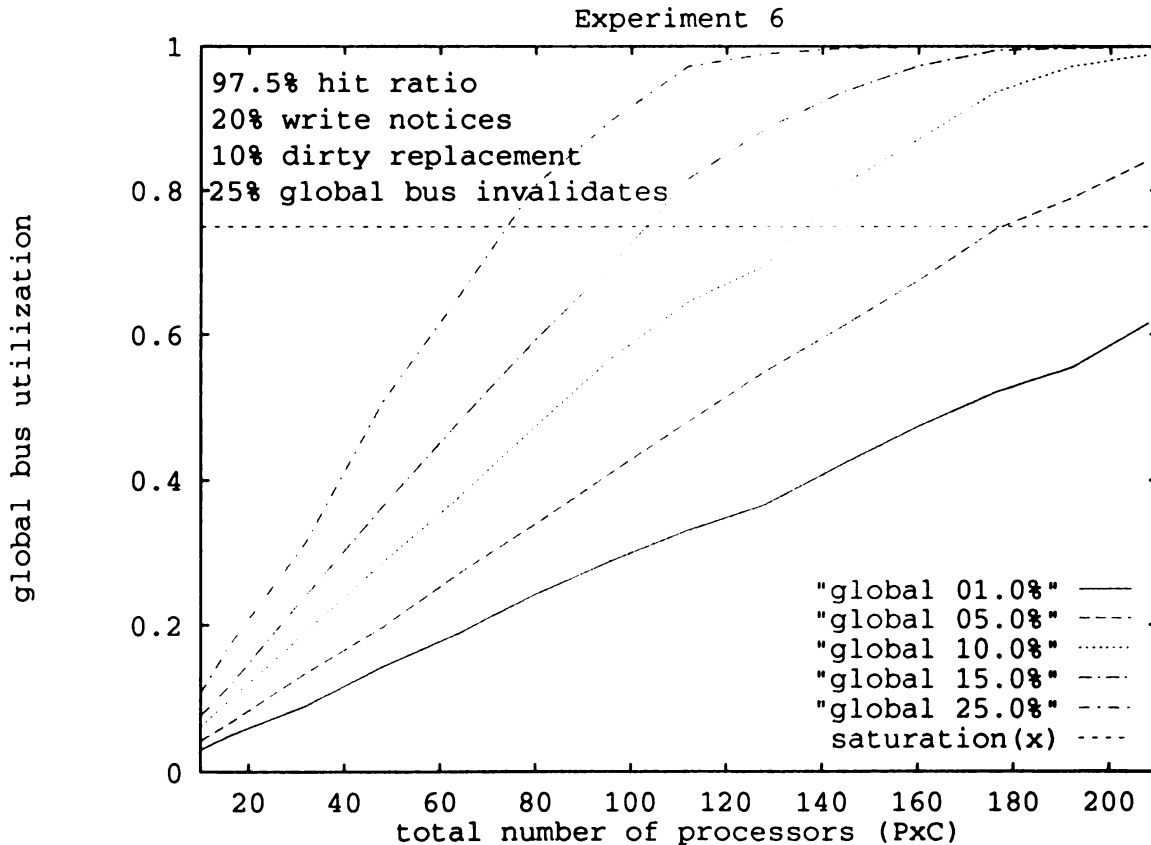


Figure 7.8: Global bus utilization for realistic application.

The effect of increased global bus use can be seen in the average MCPI tabulated in Table 7.7. The rise in MCPI is not as dramatic as global bus utilization because the individual cluster buses are not running on the edge of saturation; this buffer acts to protect MCPI until the global bus congestion is so severe that both MCPI and cluster bus utilization begin rising. Cache controllers performing transactions which require the global bus (a read request to global memory, or a global bus invalidate, for example) first arbitrate for their cluster bus, and only then request the global bus. Any queuing delay experienced at the global bus means a cache controller is also holding its cluster bus, effectively preventing any other cache from using it.

Total Processors (Clusters)	Global Memory Accesses							
	1%	2%	5%	7%	10%	15%	20%	25%
8 (1)	1.96	1.94	1.94	1.96	1.94	1.96	1.95	1.94
16 (2)	1.94	1.94	1.95	1.95	1.96	1.95	1.97	1.97
32 (4)	1.94	1.94	1.94	1.96	1.96	1.97	1.97	1.97
48 (6)	1.95	1.94	1.95	1.96	1.96	1.98	1.99	2.01
64 (6)	1.95	1.94	1.96	1.96	1.97	1.98	2.01	2.03
80 (10)	1.95	1.94	1.96	1.96	1.97	1.98	2.01	2.03
96 (12)	1.94	1.95	1.95	1.97	1.99	2.04	2.12	2.25
112 (14)	1.94	1.95	1.96	1.97	2.00	2.10	2.25	2.50
128 (16)	1.94	1.94	1.97	1.98	2.02	2.16	2.38	2.75
144 (18)	1.95	1.96	1.98	2.00	2.07	2.29	2.59	3.05
160 (20)	1.95	1.95	1.99	2.03	2.10	2.42	2.88	3.53
176 (22)	1.95	1.96	2.00	2.05	2.21	2.71	3.24	3.77
192 (24)	1.95	1.97	2.03	2.09	2.36	2.92	3.48	4.11
208 (26)	1.96	1.97	2.05	2.18	2.48	3.11	3.79	4.39

Table 7.7: Multiprocessor cycles per instruction (MCPI) for non-ideal application Experiment 6.

As expected for this less-than-ideal workload, Figure 7.9 shows the plot of potential efficiency decreasing significantly for even low levels of global access. Not only does

efficiency fall off more rapidly for this simulated application (when compared to the last experiment), but the maximum efficiency possible is also lower. For 10% global access levels the last experiment had a potential efficiency of 82.5% and a maximum theoretical speed-up of 172. By comparison, at 10% global access for this experiment, a maximum size configuration of 208 processors has a potential efficiency of only 64% for a maximum speed-up of 133.

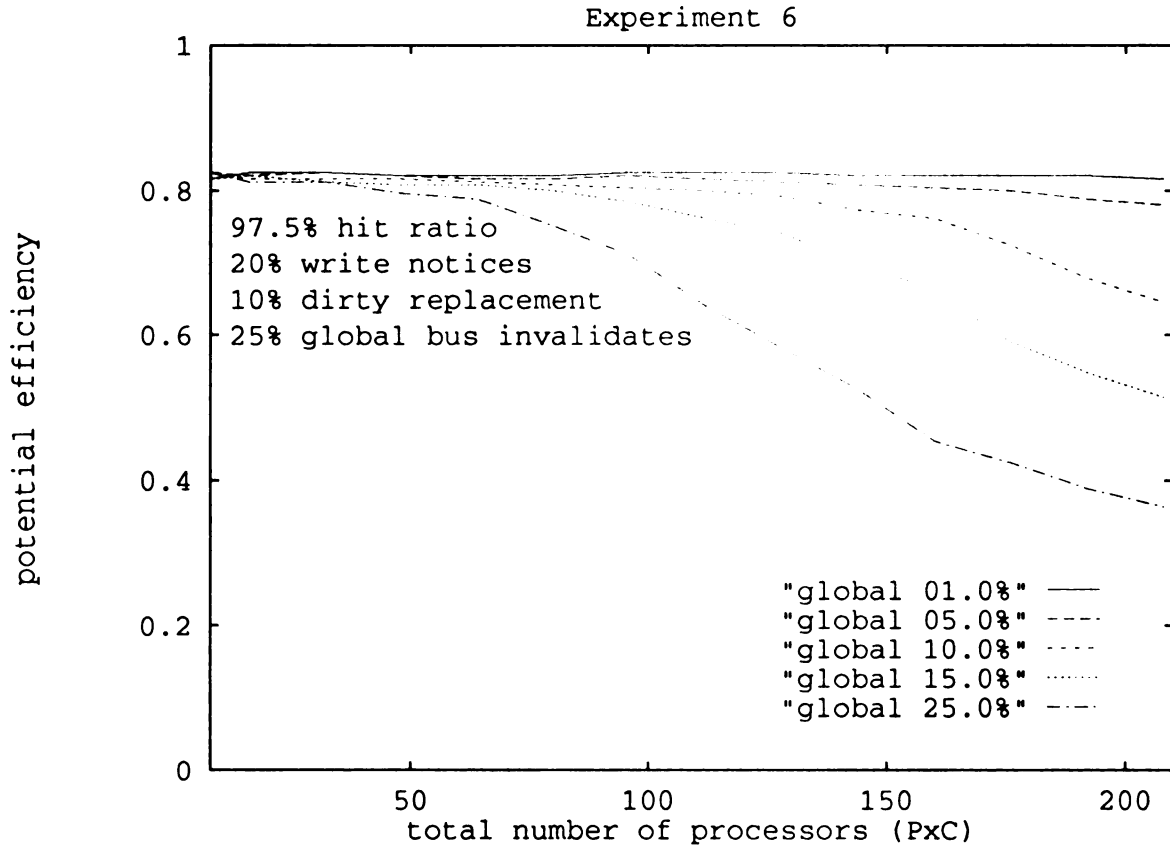


Figure 7.9: Potential efficiency for realistic application.

7.4 Trace-driven Model

With a purely trace-driven simulation it is possible to accurately model the operations of the HBSM multiprocessor; probabilistic characterization of memory reference dependent events can be avoided. A purely trace-driven model would represent the state of every cache, memory and controller in the HBSM multiprocessor. Simulation time for such models is mostly determined by the size of the trace, and to a

lesser extent on the parameters of the model (cache geometry, size of system, etc). In addition, since the number of events in a complete trace-driven model is larger than a probabilistic model, simulation time as measured in CPU seconds per memory access would be larger. Simulation of models of large systems with realistically sized traces would be both computationally and storage space intensive. A hybrid trace-driven/probabilistic model can be used as a compromise between estimating all parameters and unreasonable simulation running times and trace storage space requirements. The strategy behind the hybrid model is to accurately model a single cluster and the global bus, but to represent intercluster interactions probabilistically. This strategy can be justified by the structure of the HBSM itself; the clustered nature of the architecture means that applications which are well suited for the HBSM multiprocessor will tend to have a much greater level of intracenter than intercenter activity. Compared to a complete trace-driven model, simulation times will be proportional only to the number of processors per cluster, and will be independent of the number of clusters studied. Intercenter activity can be characterized by studying the trace of the application being simulated, thus reducing the uncertainty of the probabilistic representation.

An additional simplification of the hybrid model is to model separate instruction caches probabilistically. Data caches are fully represented, so the hit rate of a data cache is determined by the cache's geometry, replacement policy, and the memory reference patterns in the traces. Instruction caches, on the other hand, are represented in the model by a single number: hit ratio. The performance results (MCPI, potential efficiency, bus utilization) from the hybrid model contain the effect of instruction fetches, since instruction misses are handled by the cache controller in the same manner data misses are, but actual address traces are not used for instructions. Taking a more abstract approach to modeling instruction fetches is a reasonable method of reducing simulation complexity, since instruction caches have higher hit ratios and better overall performance than data caches, and they do not generally have dirty blocks to replace. In addition, since the number of instructions fetched is typically two to three times greater than the number of data accesses, the decrease in simulation time is significant.

The application used in the trace-driven simulations is a parallel merge sort. The algorithm for the parallel sort attempts to balance the amount of work done in each processor over all phases of execution and has been proposed as a shared memory

multiprocessor benchmark. The parallel sort is an iterative, barrier synchronized algorithm built on an efficient, asynchronous parallel merge algorithm capable of effectively using P processors to merge two ordered arrays of size $N/2$ into a single ordered array of size N [24]. The performance of the HBSM multiprocessor on the parallel merge sort trace of 8192 random, integer data elements done by an 8 cluster, 8 processors per cluster multiprocessor is used to gauge the accuracy of the probabilistic model.

The parallel merge sort appears to be a reasonably good match to the HBSM multiprocessor architecture, since the list of data to be sorted can be stored in global memory, and each processor accesses mostly its own cluster memory. Since processors work in groups of progressively larger powers of two, when the size of the groups exceeds the number of processors per cluster, intercluster sharing of global data is done for both reading and writing. Sorting is an important application of computers, and the parallel merge sort lies in the middle of the spectrum of algorithms suited for the hierarchical bus architecture. It is thus reasonable to hope that high performance of the HBSM multiprocessor memory subsystem on the parallel merge sort is indicative of potentially high performance on other parallel applications.

The performance of the simulation model, as measured by the number of memory references (both data and instruction) per second of CPU time on an otherwise unloaded 25MHz 68040 NeXT workstation is only 850 references/second. This compares to 3800 references/second for the probabilistic model with comparable cache hit ratio, and 60 references/second for the highly detailed Petri net model. The difference in speed is due to the difference in detail of the two models; the probabilistic model consists of 32 events in 760 lines of C++ , versus 51 events in 1970 lines of C++ for the hybrid model. The performance of the hybrid model is only weakly dependent on the characteristics of the model being simulated compared to the probabilistic model. This is a result of both the detailed level of modeling and the time spent in disk I/O for the trace-driven model.

7.4.1 Intercluster activity

The hybrid model accurately represents the state of a single cluster, the global bus and memory. Assuming that the behavior of the traced application in the full HBSM multiprocessor is symmetrical with respect to clusters (true for the parallel merge

sort), the actions and performance of one cluster will be representative of all clusters. For the hybrid model the traces are used to estimate the level of intercluster activity. The measurement made to gauge intercluster activity is the average number of clusters from which a global block is accessed. While a single measurement could be used to represent this average, the synchronized loops of the parallel merge sort offer an opportunity to increase the resolution of this measurement. The number of clusters accessing each global block is averaged across blocks for each synchronization period to determine A_c . As expected, the level of sharing as measured by A_c increases steadily with each synchronization period of the parallel sort. Only blocks which are accessed by at least one cluster (that is at least one processor on one cluster) are counted in the measurement of A_c . This means that A_c ranges from 1.0 to C , where C is the number of clusters. If blocks were only accessed by one cluster A_c would be 1.0. If blocks were accessed by all clusters A_c would be C . From A_c the following probability can be derived:

$$\Pr(\text{block } b \text{ is held in cluster } c) = \frac{A_c}{C}$$

This is actually a conditional probability, since only blocks which are accessed by at least one cluster are used to measure A_c .

The COGI cache coherence protocol requires that 8 types of intercluster activity take place. These events are shown in Table 7.8. Some simplifying assumptions reduce the number of these intercluster events which must be modeled probabilistically. First, if each processor is limited to accessing local cluster memory and global memory, then the CMC will not be required to act in cases 1 through 4. This restriction is reasonable, since widely shared data can be placed in global memory, and cluster memory can be reserved for processor private memory. The traced parallel merge sort application uses memory in this way. Second, if the CCC has a very large, highly associative cache, then case 8 can be ignored. Since the CCC only caches status, and not data, its impact on performance of the system will not be as important as a traditional second level cache. In addition, the CCC can be made very large, for a fixed number of transistors, as compared to a traditional data/status cache, since status information requires only a fraction of the storage that status and data require. The remaining cases (5, 6, and 7) are the results of read requests or write notices to global blocks that may be held in multiple clusters.

Case	Controller	Transaction	Action taken
1	CMC gsnoop	global bus RR	cluster bus RR
2	CMC gsnoop	global bus WB	cluster bus WB
3	CMC csnoop	cluster bus RR	global bus RR
4	CMC csnoop	cluster bus WB	global bus WB
5	CCC gsnoop	global bus RR	cluster bus flush
6	CCC gsnoop	global bus IN	cluster bus IN
7	CCC csnoop	cluster bus WN	global bus IN
8	CCC csnoop	replacement policy	cluster bus IN

Table 7.8: Cases of intercluster transactions in COGI protocol.

The CCC sends a global bus invalidate when it observes a cluster bus write notice to a block which may be shared by other clusters. In the typical case of no intercluster write sharing, a global bus invalidate would be broadcast for the first write notice to a block, and further write notices for this block would remain in the cluster. Case 7 of Table 7.8 can thus be modeled quite accurately for low levels of write sharing just by following the COGI protocol. Traffic on the global bus due to invalidates from other clusters (those not being modeled) is assumed to occur at the same rate as the cluster being modeled generates GBINs. Every GBIN broadcast from the modeled cluster creates $C - 1$ other GBIN transactions on the global bus.

For every invalidate on the global bus, the CCC gsnoop of each cluster decides whether a cluster bus invalidate is required on its cluster. The measured trace characteristic A_c can be used to determine the likelihood of these case 6 transactions. Assuming symmetry between clusters, the number of global bus invalidates broadcast by each cluster should be approximately the same. Thus, the cluster being simulated will see the global bus invalidate traffic from $C - 1$ other clusters. The probability that a global bus invalidate from cluster Y is translated into a cluster bus invalidate by the CCC on cluster X is

$$\begin{aligned}
 \Pr(\text{CBIN in cluster X}) &= \Pr(\text{block in X} \ \&\& \ \text{block in Y}) \\
 &= \Pr(\text{block in X} \mid \text{block in Y}) \times \Pr(\text{block in Y})
 \end{aligned}$$

by the definition of conditional probability. Since A_c accounts for only those blocks held by at least one cluster (*i.e.* it is never less than 1.0), the probability of a block residing in one of the other clusters (X) not responsible for generating the global bus invalidate (Y) is

$$\Pr(\text{block in X} \mid \text{block in Y}) = \frac{A_c - 1}{C - 1}$$

so that

$$\Pr(\text{CBIN in cluster X}) = \frac{A_c - 1}{C - 1} \times \frac{A_c}{C}$$

When blocks are held by only one cluster at a time ($A_c = 1$, no sharing), the probability of a CBIN is 0, as expected. When blocks are held by all clusters at all times ($A_c = C$, maximal sharing), the probability of a CBIN is 1.0, as expected.

Using this probability, and the assumption of symmetry between clusters, the model translates a GBIN from one of the $C - 1$ other clusters with the above probability. This will correspond to a rate of

$$(C - 1) \times \frac{A_c - 1}{C - 1} \times \frac{A_c}{C} = \frac{(A_c - 1)A_c}{C}$$

CBIN events for the cluster being simulated for every GBIN generated on the simulated cluster. The accuracy of determining CBIN activity from GBIN activity is only as good as the accuracy of the GBIN activity model. If a block is being heavily shared between two clusters, and one of the clusters writes to it, then it is likely that a global bus read request from the invalidated cluster will soon come along and reset the state of the block in the first cluster to shared, so that further writes in this cluster will result in global bus invalidates. This corresponds to case 5 in Table 7.8. Because A_c does not distinguish between the type of access performed (read or write), and because the relative timing of events in different clusters is impossible to measure accurately from the traces, the rate of GBINs generated (and hence of CBINs) will be smaller than those which would take place in the actual multiprocessor. The flush transactions which determine this inaccuracy (case 5) are not modeled at all.

7.4.2 Benchmark parallel merge sort

The parallel sort algorithm divides a data set into P segments, uses a single processor and a sequential sorting algorithm (Quicksort, say) to sort each segment, then applies

the effective parallel merge to iteratively merge the resulting P sorted segments into a single sorted data set. The parallel merging requires $\lceil \log_2 P \rceil$ iterations to fully merge the original P data segments. Figure 7.10 shows the parallel merge sort algorithm. The original data set \mathcal{S} is first divided into P segments each denoted by K , where $1 \leq K \leq P$. The functions $\text{ub}(\mathcal{X})$ and $\text{lb}(\mathcal{X})$ return the upper and lower bound of their array arguments, respectively. The values $\text{lo}(\mathcal{X}, K)$ and $\text{hi}(\mathcal{X}, K)$ represent the beginning and ending indices of array \mathcal{X} demarking the region of \mathcal{X} in which the K^{th} processor operates. In each iteration of the loop in Figure 7.10 teams of processors work together to merge two segments of \mathcal{S} (labeled \mathcal{A} and \mathcal{B}) into a segment of the destination array \mathcal{D} (labeled \mathcal{C}). In the first iteration, teams of two processors merge two segments of length $|\mathcal{S}|/P$ from \mathcal{S} into \mathcal{D} . In the second iteration, teams of four processors merge two segments of length $2 \times (|\mathcal{S}|/P)$ from \mathcal{S} into \mathcal{D} . The final iteration uses all processors to merge two segments of length $|\mathcal{S}|/2$ into the destination \mathcal{D} . The details of the balanced parallel merge can be found in [24].

7.4.3 Results for 8×8 system

The trace-driven model requires the specification of cache size, block size and cache associativity. Cache hit ratio will depend on all of these factors. The simulations done for this chapter were based on 454,193 data references (read and writes) and 1,495,464 instruction fetches for a single cluster of 8 processors (243,707 memory references per processor). The level of sharing (accesses to global blocks, as defined for the probabilistic model) measured in the traces is 13%. Both cache and block sizes are reported in units of 32-bit words. The parallel sort was done on random integer data ranging from 0 to 8,192,000. Each element of the data to be sorted required a single 32-bit word for storage, thus a 256 word cache can store 256 data elements. For experiments in which the geometry of the cache (set or block size) is varied, the size of the cache is held constant. In other words, a cache of 1024 words with a block size of 1 can hold 1024 blocks, while the same cache with a block size of 16 can hold only 64 blocks.

The first experiment with the trace-driven simulation model fixes the associativity and block size at typical values in order to study the performance of the system for various cache sizes. The data plotted in Figure 7.11 are cluster and global bus utilizations as a function of cache size for a 2-way set associative cache and a block

Let

\mathcal{S} be the array of data to be sorted
 \mathcal{A} be a segment of \mathcal{S} worked on by a group
 \mathcal{B} be a segment of \mathcal{S} worked on by a group
 \mathcal{D} be the destination array
 \mathcal{C} be a segment of \mathcal{D} to store the merge of \mathcal{A} and \mathcal{B}
 p be the size of a group of processors
 k be the number of the K^{th} processor within a group

Split \mathcal{S} into P segments labeled with K

$\text{lo}(\mathcal{S}, K) = \text{lb}(\mathcal{S}) + (K - 1) \times [\text{ub}(\mathcal{S}) - \text{lb}(\mathcal{S}) + 1] / P$
 $\text{hi}(\mathcal{S}, K) = \text{lo}(\mathcal{S}, K + 1) - 1$

Quicksort the P segments

For $i = 1$ to $\lceil \log_2 P \rceil$ in barrier synchronized loops do

$p_K = \min(2^i, P - 2^i \times (\lceil K/2^i \rceil - 1))$

$k_K = ((K - 1) \bmod 2^i) + 1$

$\text{lb}(\mathcal{A}_K, i) = \text{lo}(\mathcal{S}, 2^i \lfloor (K - 1)/2^i \rfloor + 1)$

$\text{ub}(\mathcal{A}_K, i) = \text{hi}(\mathcal{S}, \min(P, 2^i \lfloor (K - 1)/2^i \rfloor + 2^{i-1}))$

$\text{lb}(\mathcal{B}_K, i) = \text{lo}(\mathcal{S}, \min(P, 2^i \lceil K/2^i \rceil - 2^{i-1} + 1))$

$\text{ub}(\mathcal{B}_K, i) = \text{hi}(\mathcal{S}, \min(P, 2^i \lceil K/2^i \rceil))$

$\text{lb}(\mathcal{C}_K, i) = \text{lo}(\mathcal{D}, 2^i \lfloor (K - 1)/2^i \rfloor + 1)$

$\text{ub}(\mathcal{C}_K, i) = \text{hi}(\mathcal{D}, \min(P, 2^i \lceil K/2^i \rceil))$

Perform balanced parallel merge using $p, k, \mathcal{A}, \mathcal{B}, \mathcal{C}$

Figure 7.10: The benchmark parallel merge sort used for traces in the hybrid model.

size of 4 words. Figure 7.12 shows the potential efficiency and the cache hit ratio for this same system. In order to achieve a 97.5% cache hit ratio, the cache must be 256 words in size. For this cache size, the cluster and global bus utilization are 58% and 80%, respectively. The purely probabilistic model predicted a cluster bus utilization of 70% for an 8×8 multiprocessor at 97.5% data cache hit ratio, and a global bus utilization of 48%, as shown in Figure 7.8. Thus, for the same data cache hit ratio, the trace-driven model gives performance estimates which are slightly better for cluster bus utilization and significantly worse for global bus utilization. As expected, the high bus utilization results in a lower value of potential efficiency; at a cache size of 256, each instruction takes 2.1 times longer to complete than the theoretically possible MCPI, and the maximum potential efficiency of the system is only 47%.

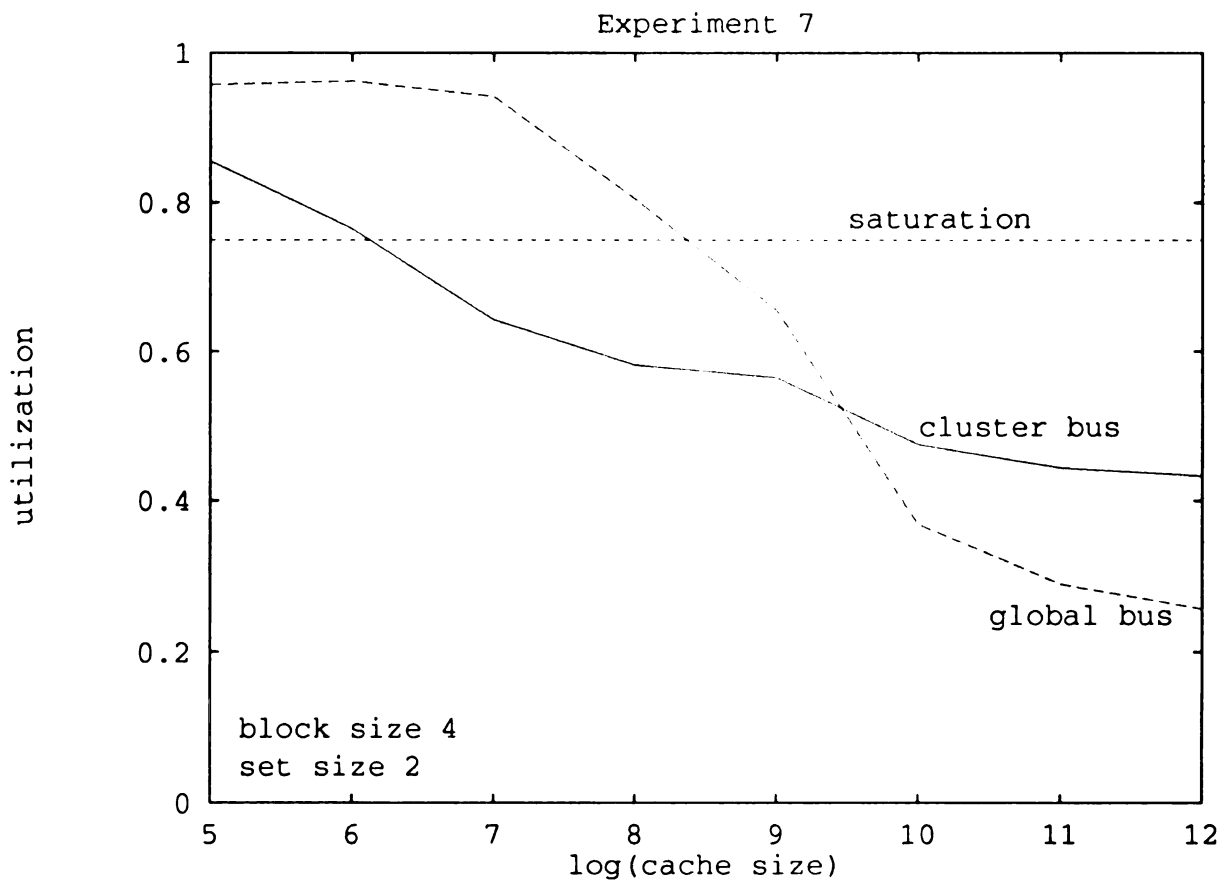


Figure 7.11: Bus utilizations in typical configuration of block size 4, set size 2.

The explanation of the discrepancy between the two models can be found by examining the nature of the memory reference patterns for the parallel merge sort.

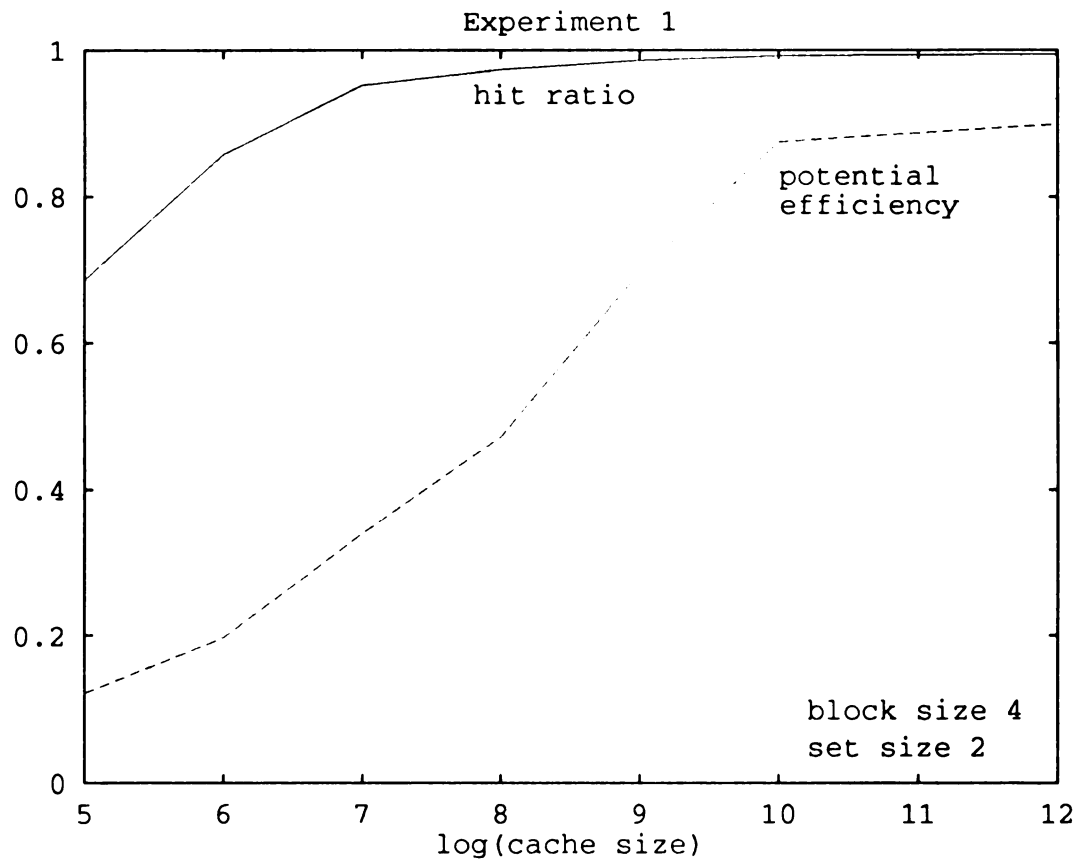


Figure 7.12: Hit ratio and potential efficiency in typical configuration of block size 4, set size 2.

Each processor in the parallel merge makes memory accesses to global memory (13%) and to processor private cluster memory (87%). The cache behavior of these two groups is very different. For processor private memory the cache hit ratio is 98.6%. For global memory the cache hit ratio is only 89.2%. While some (3.5%) of the cache misses for global blocks are serviced by other caches on the cluster which have dirty copies of the block, the remainder of the misses to global blocks are serviced by the global memory. Access to global memory is slower than cluster memory because of the time required to transfer the memory block across two buses, and because of the contention delay for two buses.

The two other factors contributing to the higher-than-expected global bus utilization are the percentage of replacement of dirty blocks and the likelihood of global bus invalidations. The experiments for the purely probabilistic model examined these two parameters over a range of 1 to 25%. For cache size of 256 (97.5% hit ratio), the replacement of dirty blocks on cache misses occurs 43% of the time. Of these dirty replacements, nearly half are global blocks which require lengthier write back transactions.

The second factor is the global bus invalidation traffic. The percentage of cluster bus write notices that result in global bus invalidate transactions is 41%. This high level of invalidates is closely related to the large number of dirty global blocks that are selected for replacement. Each dirty block that is replaced has a high likelihood of being accessed again by the same cache. Every time a block is written back to the global memory and read into the cache, a subsequent write will require a global bus invalidate, since global blocks can not be held in a clean, cluster exclusive state.

In order to avoid memory system performance degradation (as judged by potential efficiency) in the typical³ 8×8 system, cache size must be increased to 1024 words. Figure 7.12 shows that at this cache size the high data cache hit ratio attained (99.2%) increases the potential efficiency to 87.5%. Large caches and subsequently high cache hit ratios also result in more reasonable values for dirty block replacement (20%) and for global bus invalidates (17%). In turn, a side effect of less dirty block replacement is more intracluster dirty sharing between caches. The probability that a cache supplies a dirty block upon a cluster bus read request is 30% for large cache sizes compared to only 3.5% for a cache of 256 words. Since efficient cache-to-cache transfer of dirty blocks was one of the design goals of the COGI protocol, increased levels of dirty

³Block size of 4, set associativity of 2.

sharing help performance of the whole system. As a result, the cluster and global bus utilizations in Figure 7.11 are well below the saturation point at 48% and 36%, respectively.

7.4.4 Cache geometry

Two factors determine the geometry of a cache: block size and set size. Block size refers to the number of words which comprise the smallest unit of coherence in the system. Larger blocks require more transfer time on the bus, but may benefit the cache hit ratio by pre-loading words adjacent to an accessed word. On the other hand, smaller blocks allow better use of the cache space, since they decrease the number of words that are loaded in the cache and never accessed. The associativity of a cache refers to the number of cache frames in which a given block may be stored in the cache. Fully associative caches allow any block to be stored in any frame of the cache. Direct mapped caches (associativity of one) specify one particular cache frame for storing each block. A compromise organization (in both cost and performance) is a set associative cache; each block of memory can be stored in one of S cache frames.

The effect of block size on the performance of the HBSM multiprocessor was studied with the trace-driven simulation model by comparing the performance of block sizes from 1 word to 16 words for 2-way set associative caches. Table 7.9 shows the MCPI data for this configuration. Figure 7.13 shows the plot of potential efficiency as a function of block size.

Examination of Table 7.9 and Figure 7.13 reveals that for reasonable sized caches (greater than 256) the best overall system performance is achieved for a block size of four. Smaller caches show a distinct peak in the curves of potential efficiency versus block size, while larger caches exhibit the same behavior but less dramatically. The point at which performance begins to degrade with increasing block size is known as the memory pollution point [22]. Large block size has both a positive pre-loading effect and a negative space-wasting effect on hit ratio. For small caches the space wasted with large blocks (wasted because some words of a block may never be accessed), is relatively more important than it is in large caches, so the performance of the system past the memory pollution point degrades rapidly.

As expected from uniprocessor experience with caches, the performance of the system is better for larger set size. Figure 7.14 shows the positive benefit that larger

Cache Size	Block Size				
	1	2	4	8	16
32	8.86	8.50	11.04	16.62	24.59
64	7.57	5.74	6.76	10.32	17.68
128	6.33	4.34	3.93	5.73	9.61
256	4.95	3.41	2.82	3.22	4.82
512	3.27	2.34	1.91	1.94	2.66
1024	2.00	1.62	1.53	1.54	1.65
2048	1.83	1.54	1.50	1.51	1.59
4096	1.73	1.52	1.48	1.49	1.56

Table 7.9: Experiment 7 multiprocessor cycles per instruction (MCPI) data for set size 2.

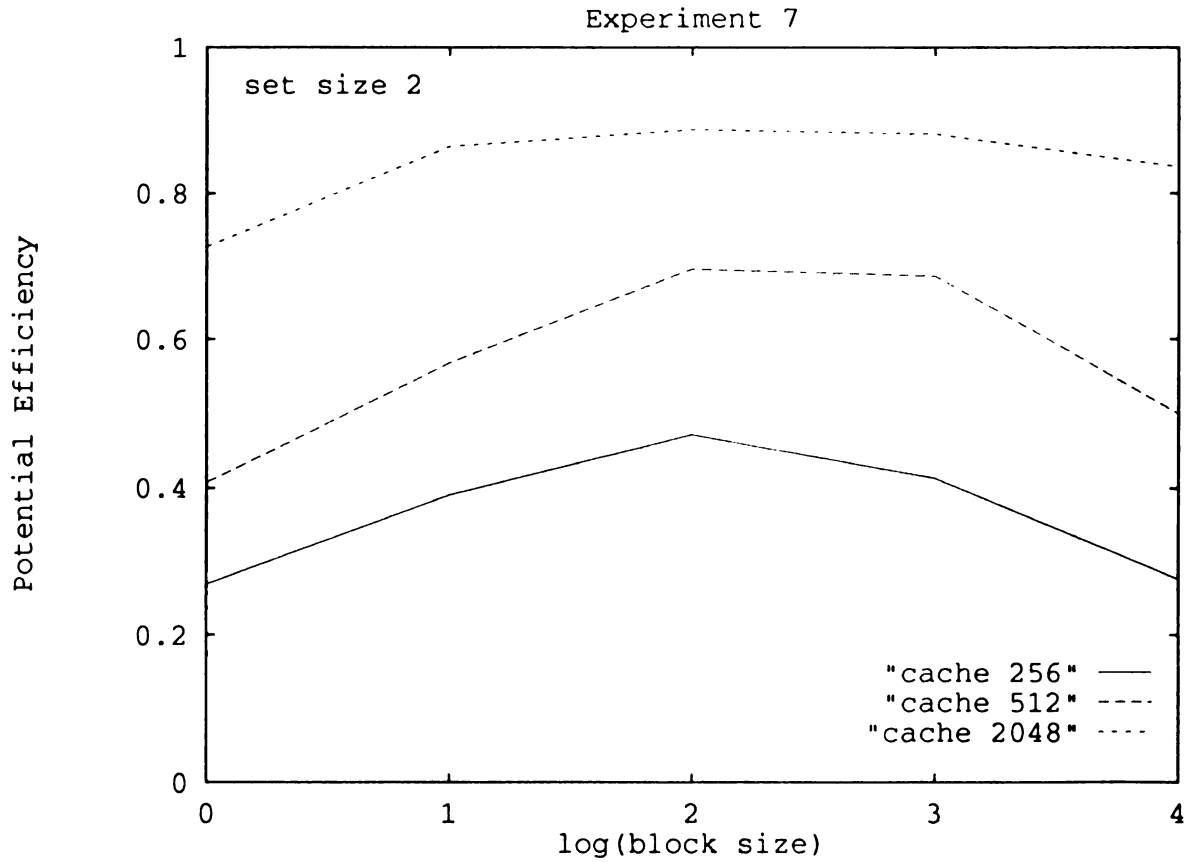


Figure 7.13: Effect of block size on potential efficiency for set size 2.

set size has on performance, particularly for small caches. For larger caches the gain in performance from a set size of 2 to 4 probably does not justify the additional expense of the higher associativity cache.

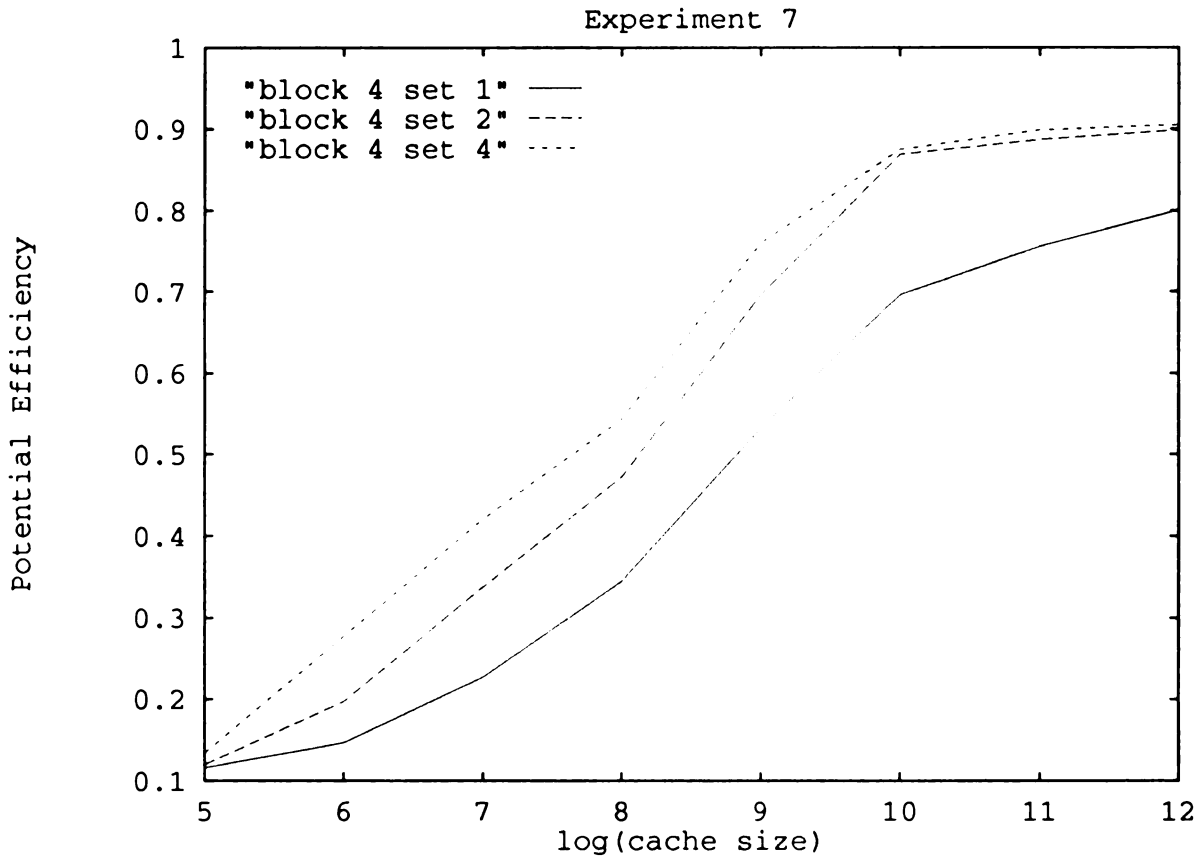


Figure 7.14: Effect of set size on potential efficiency for optimal block size.

The effect of set and cache size on global bus utilization is shown in Figure 7.15. For reasonable sized caches, the large improvement in global bus utilization gained from a 2- or 4-way set associative cache indicates that the global memory blocks accessed by a processor tend to interfere with each other in the cache (*i.e.* they map to the same cache frame). Increasing the set size decreases misses due to this interference, lowering the global bus utilization.

7.4.5 Coherence traffic

An advantage of the trace-driven simulation is the ability to accurately characterize the impact of coherence activity on the system. The results in this section are taken from the trace-driven simulation of the 8×8 multiprocessor sorting 8192 random

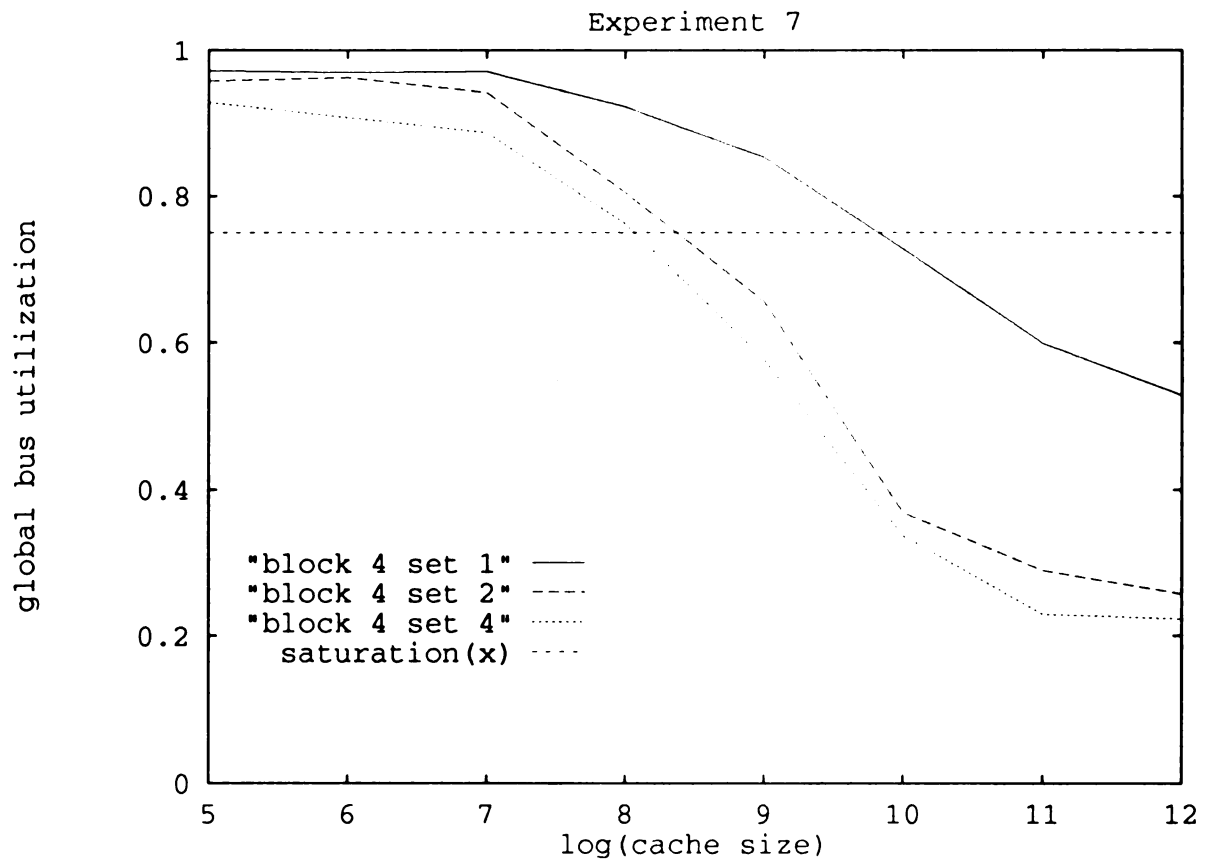


Figure 7.15: Effect of set size on global bus utilization for block size 4.

integers. The cache size is fixed at 1024 words, with resulting cache hit ratios in the neighborhood of 96–99%.

Write notices

The number of write notices placed on the bus by the CC cache controllers decreases as block size increases, as shown in Figure 7.16. Since a write to a single word in a block will place the block in a private, dirty state, large block size reduces cluster bus write notice traffic for blocks which are not being shared for writing between two or more processors.

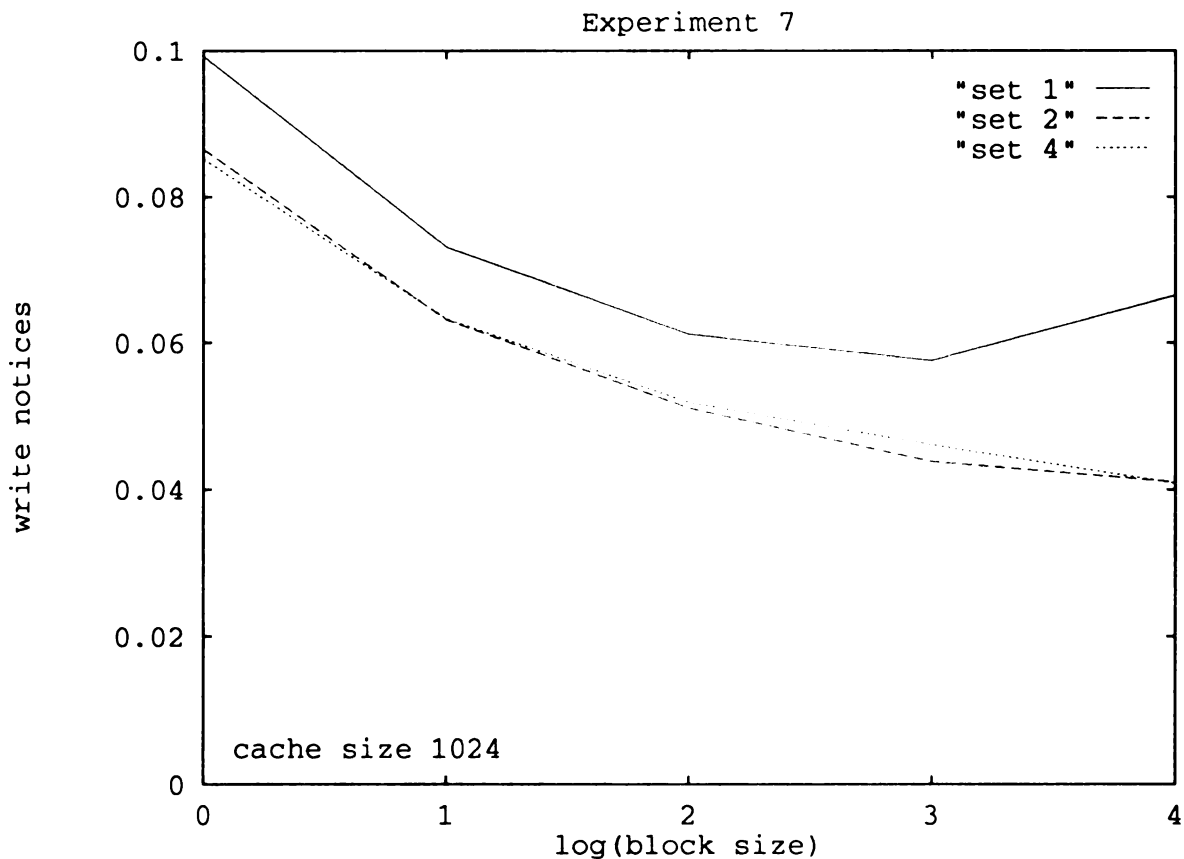


Figure 7.16: Fraction of cache writes which generate a cluster bus write notice.

As expected, the fraction of write notices issued for blocks that are not actually shared by other caches on the cluster drops significantly with increasing block size. For large block sizes in 2- and 4-way associative caches the unnecessary cluster bus write notice traffic is minimal, as shown in Figure 7.17. With a block size of 16, a single write potentially replaces 15 unnecessary write notices that may occur for

a block size of one. Write notice traffic for blocks which are being actively shared between processors is not affected very strongly by block size. The advantage of large block sizes is seen for private memory. Reducing the number of unnecessary write notices helps to decrease bus utilizations.

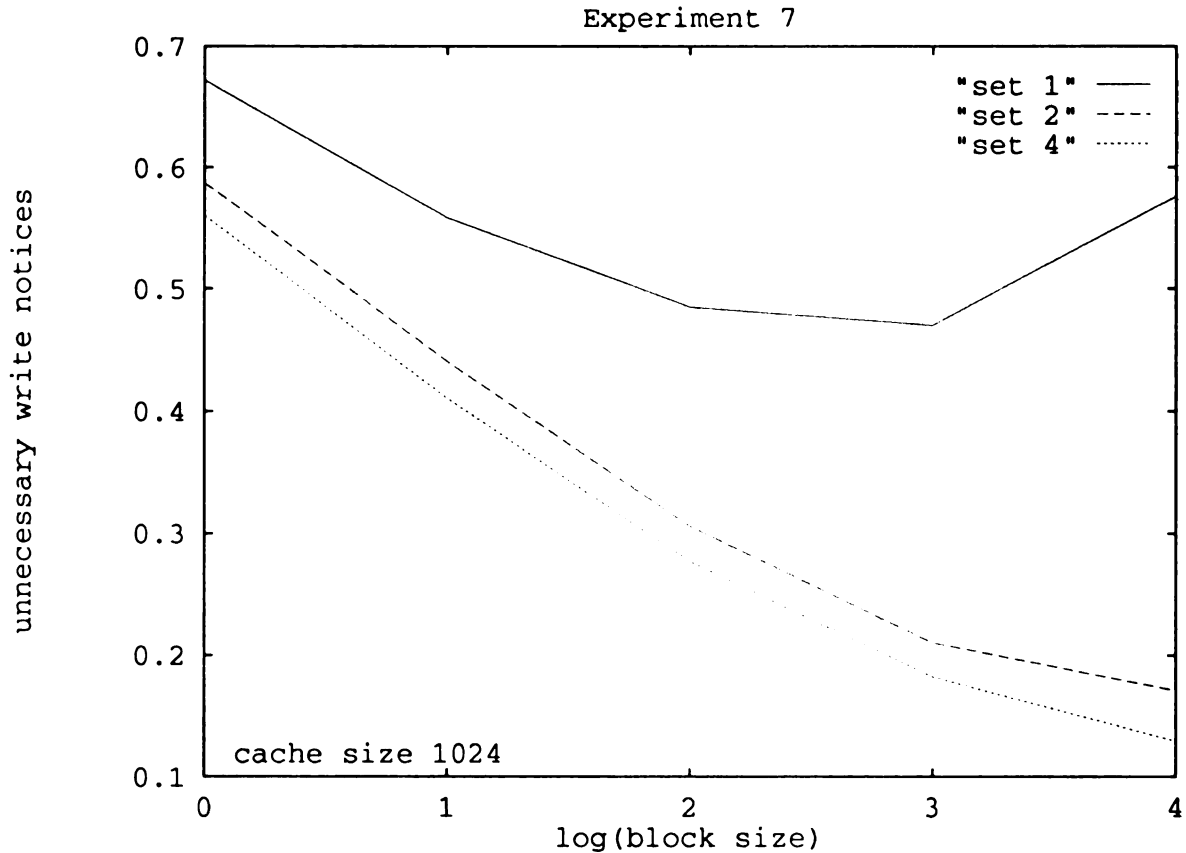


Figure 7.17: Fraction of write notices to blocks which are not shared by other caches.

Dirty sharing

Dirty sharing takes place between CC caches for global memory blocks. Figure 7.18 shows the fraction of read misses which are serviced by another cache on the cluster with a dirty copy of the block. Cache-to-cache transfers of dirty blocks are faster than reading blocks from memory, so the decrease of sharing for large block sizes increases bus utilizations. For set associative caches this effect is very small, and is probably due to large block size decreasing the time that dirty blocks remain in caches without being replaced.

The readily apparent differences between direct-mapped caches (associativity 1)

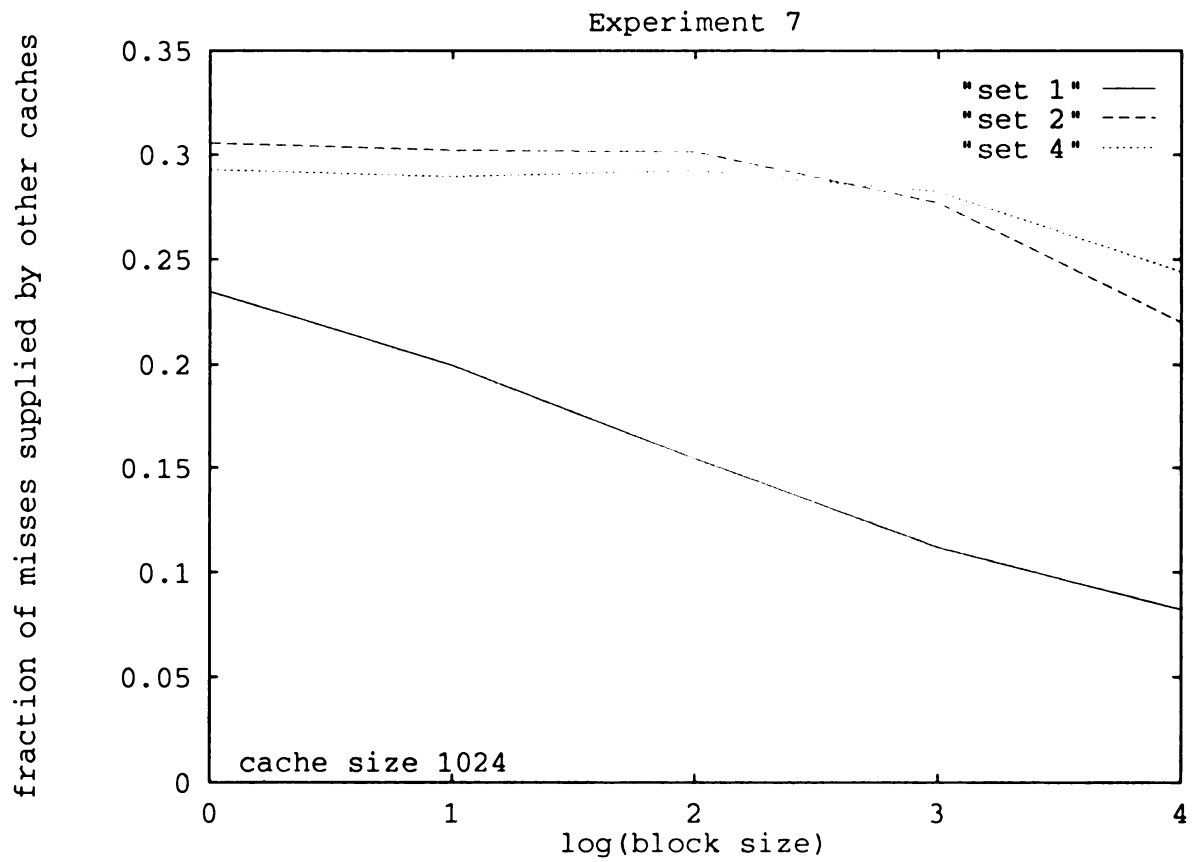


Figure 7.18: Fraction of read misses which are serviced by another cache with a dirty copy of the block.

and set associative caches in Figures 7.16 7.17 and 7.18 show the advantage of increasing the flexibility of the cache block replacement algorithm. The adverse impact of large block size on direct mapped caches results in a lower optimum block size (2 compared to 4) for such caches.

Bus traffic

One of the advantages to be gained from detailed trace-driven simulations is the ability to account completely for the use of each system resource. The cluster bus utilization in an 8×8 multiprocessor with cache size of 1024, block size of 4, and 2-way associative caches is 48%. Table 7.10 shows how the busy time of the bus is divided among the possible bus transactions.

Transaction	Bus Busy Time (%)
instruction fetch	73.4
global memory read	11.2
write notice	4.1
cluster memory read	3.9
cache-to-cache transfer	2.7
global write back	2.2
cluster write back	1.9
global invalidate	0.7

Table 7.10: Experiment 7 distribution of cluster bus busy time.

The large portion of time required for instruction misses indicates the importance of maintaining a high instruction cache hit ratio. The relatively small portion of time (7.5%) spent in maintaining coherence (write notices, global bus invalidates, cache-to-cache transfers) indicates that the COGI cache coherence protocol does not impose a great cost on the system to maintain coherence.

7.5 Summary

The purely probabilistic model was used to identify a target configuration for the HBSM multiprocessor. Simulations for a single cluster indicated that 8 processors per cluster could be supported with a potential efficiency of 84% at a 97.5% data cache hit ratio. Multiple cluster experiments showed that an ideal application (one with no global bus invalidations and no dirty block cache replacement) could achieve the target level of efficiency for configurations of 200 processors and global access probabilities of up to 10%. For a less ideal application (25% global bus invalidations and 10% dirty block replacement), the potential efficiency of the system with 10% global accesses degraded below the 84% target level at only 100 processors.

The experiments done with the trace-driven model show that the HBSM multiprocessor architecture can readily support 8 clusters of 8 processors each for the parallel merge sort application. The relatively high levels of global memory access (13%) and the amount of intercluster sharing indicated by the 30% or greater global bus invalidation rate and write sharing of 3 to 10% indicate that the parallel merge sort is not an overly good match for the hierarchical bus architecture. Considering the imperfect match of the parallel merge sort to the HBSM architecture, and extrapolating from the readily attainable cluster bus utilization of 45% and global bus utilization of 20% indicates a maximum scalability of 120 to 200 processors for the parallel merge sort. Judging by the limit on potential efficiency, such a system could be expected to show a speed-up on good parallel applications of 90% of the size of the multiprocessor. Applications which exhibit more clustering of memory accesses and fewer global memory references should easily reach the goal of efficiently utilizing an HBSM multiprocessor of 200 processors.

Applications which would be more suited to the HBSM multiprocessor include the large group of problems which operate in a small neighborhood of a large data set. Image processing problems are an obvious example of this sort of application. Since a very large amount of data could be distributed amongst the clusters, the only processors that would use the global bus heavily would be those that operate at the cluster-defined boundaries of the data. Users who need to make many runs of a simulation program (such as graduate students) could use the HBSM multiprocessor very efficiently, since each processor could be assigned a different set of parameters to generate one data point in a graph. Such applications would require no intercluster

sharing at all. Simulations of complex systems such as VLSI circuits may be able to be partitioned to run efficiently on a clustered architecture.

The most important variable to the overall performance of the HBSM multiprocessor is cache size. Inadequately sized caches increase bus utilization past saturation, thereby greatly increasing bus contention and queueing delay. For the 8×8 system studied in this chapter, the data cache hit ratio must be close to 99% in order to keep the potential efficiency around 90%. At 90% potential efficiency, the 64 processor multiprocessor could exhibit speed-up of 58 on a good parallel algorithm.

Given a fixed size cache, a computer designer must decide on the geometry of the cache. Set associativity is an unambiguously good thing for performance. The expense (in transistors, or silicon area) limits the associativity of today's caches to between 1 and 4. As shown in each trace-driven experiment, increasing the associativity of a cache increases the flexibility of the cache placement algorithm, so it can only help cache hit ratios. The boost in performance from using a 2-way associative cache over a direct-mapped cache is significant, while the additional benefit of a 4-way associative cache is less dramatic, particularly for large caches.

The case for block size is less clear than for set size. Increasing the block size of a fixed size cache will decrease the number of blocks which can be held in the cache. The expense (in bus busy time) of transferring large blocks across the bus is mitigated by the positive effects of pre-loading words that are likely to be accessed in the near future. In addition, the model of the memory controller used for the simulations assumed that memory was interleaved so that the time to access a block was independent of the block size. Since the time required by the memory controller to satisfy a read request is composed of the memory access and the bus transfer time, larger blocks do not take proportionally longer to retrieve from memory. The phenomenon of memory pollution, where large blocks cause wasted space in the cache, militates against larger blocks. The many interrelated factors (miss access time, cache hit ratio, write notice traffic, dirty replacement traffic, etc) which are affected by block size result in performance which is not monotonically related to block size. Trace-driven simulations indicate that the optimum block size is four words. This value does not seem to be peculiar to the parallel merge sort being studied, since global data (the list being sorted) consists of 8192 words, and there is nothing special about groups of four elements. Similar results have been found in other studies of bus-based multiprocessors [22].

The disparity between the trace-driven model and the probabilistic model shows the importance of trace-driven simulation. The probabilistic model uses an average cache hit ratio to represent the behavior of the cache. When the performance of the cache differs for different classes of memory this is an inadequate representation. Models of non-uniform memory access multiprocessors are vulnerable to this type of error, since memory access costs depend on the physical location of the memory block.

Chapter 8

Conclusions and Future Research

The goal of this work was to design and evaluate the memory subsystem for a hierarchical bus shared memory multiprocessor. The existing infrastructure of computer engineering knowledge of bus interconnects is considerable. This fact, coupled with the ability to easily expand bus-based systems in small increments, and compatibility with the shared memory programming paradigm, makes bus-based multiprocessors an important commercial computer architecture. Extending the scalability of bus-based multiprocessors past the limits of a single bus will be an important step in the eventual ascendancy of parallel processing.

The first task in the study of the hierarchical bus architecture was to broadly partition the responsibilities of the memory subsystem between three controllers, the CC, CMC, and CCC. Once the primary role of the controllers was decided upon, a protocol to maintain cache and memory coherence was developed. Detailed modeling and trace-driven simulation of the protocol and controllers was performed to assess their feasibility and correctness. The final step was to evaluate the performance of the HBSM multiprocessor in order to judge the scalability of the hierarchical bus architecture. This chapter outlines the results and conclusions of this study, and considers future work to extend some of the tools developed and improve the performance of the architecture.

8.1 HBSM Multiprocessor Architecture

The primary bottleneck which limits the ultimate scalability of bus-based multiprocessors is interconnection bandwidth. Cache memory has traditionally been used to decrease memory latency. In addition, caches in bus-based multiprocessors are necessary to decrease the demands placed on the bus by each processor. A two-level hierarchical structure with clusters of single bus multiprocessors connected via a global bus has the potential of increasing the overall size of a bus-based multiprocessor. Private caches for each processor and physically distributed memory are key

elements of the HBSM multiprocessor architecture. The use of caches and physically distributed memory becomes complicated in a hierarchical bus system because of the cache coherence problem.

The COGI cache coherence protocol has been designed to efficiently maintain the coherence of the HBSM multiprocessor cache and memory modules. COGI is a write-back/write-update protocol on the cluster buses, and a simpler write-back/write-invalidate protocol on the global bus. Important characteristics of earlier protocols have been included in COGI, such as the shared line, which allows dirty sharing of blocks on a cluster. In addition, several novel features such as data-less second level caches and support for distributed memory are central to the COGI protocol. The COGI protocol supports a single, coherent, shared address space for all processors in the system.

The simulations of Chapters 7 and 7 indicate the ultimate scalability of the HBSM multiprocessor is nearly an order of magnitude greater than currently available single bus multiprocessors. A maximum configuration of 20 clusters of 8 processors per cluster would result in a multiprocessor with a raw power of nearly 4000 MIPS¹. With large enough caches this machine could operate at 85% potential efficiency, so that perfect parallel applications could utilize 3400 MIPS of computational power.

The limit of 8-10 processors per cluster in the HBSM multiprocessor is only about half that of the currently available machines from Encore and Sequent. In part this is due to the characteristics of the processors which were simulated. A 50MHz RISC microprocessor will demand much higher memory bandwidth than the processors used in existing commercial machines. Further studies on the effects of increasing bus bandwidth, particularly the benefits of unequal distribution of resources between the global bus and the cluster buses, would be interesting. The configuration of the HBSM multiprocessor and the allocation of bandwidth resources should depend strongly on the memory access patterns of the applications of interest. A general method to characterize the memory access patterns of a wide variety of scientific applications would be very useful.

The HBSM multiprocessor, built from standard microprocessors, commodity memories and conventional buses, would be an economical means of providing significant computational power. Simulation indicates that the performance of this architecture

¹Based on a 50MHz RISC microprocessor being rated at 25 MIPS.

is heavily dependent on the level of access to global memory. Large applications running on an 20×8 machine should have no more than 10% global memory accesses in order to avoid saturation of the global bus. In addition to single large scientific parallel applications, transaction processing and time-sharing workloads would be other suitable uses for the HBSM multiprocessor.

8.1.1 Performance improvements

To satisfy the requirements of sequential consistency, the COGI protocol requires that processors and controllers be held until a transaction is completely finished. For example, a processor is not released on a write to a shared block until the CC has made a write notice transaction on the cluster bus. If this write notice should require that an invalidation be broadcast on the global bus, then the writing processor, the CC and the cluster bus will all be busy until the global bus invalidate has been accomplished. Important speed improvements may be made by allowing a weaker consistency model. Removing the restriction of sequential consistency requires that more attention be paid to explicit synchronization points, but promises to reduce utilization of system buses. The issue of the semantics of memory operations in shared memory multiprocessors is still being hotly debated.

Along with weaker consistency, using a split transaction protocol on the cluster buses would increase the number of processors that a single bus could support. As modeled, the cluster bus is held for the duration of a bus transaction. If this transaction involves memory, then the bus is actually idle, but unavailable to other processors, while the memory access is performed. A split transaction bus protocol allows for a bus master to initiate a memory access request then relinquish the bus so it may be used by other masters. When the memory access is finished, the memory module initiates a response to pass the data to the requesting processor. Split transaction protocols require that memory modules be smart enough to initiate bus transactions. (rather than simply accepting reads and writes). Cache coherence is also significantly complicated by split transaction protocols, since transactions have two phases (initiation and completion) and may be in progress when other relevant transactions occur. Interleaving transactions in this manner increases the complexity of controllers, and must be weighed against the improvement in performance. For the HBSM multiprocessor and the parallel merge sort, the distribution of busy time of

the cluster bus described in Table 7.10 indicates a maximum potential savings of 30% on cluster bus utilization. In this case, a split transaction protocol would reduce the cluster bus utilization from 48% to 18%. Increased arbitration and transaction overhead would erode the benefits from a more sophisticated protocol. A careful study of the benefits gained versus the increased complexity of the controllers and cache coherence protocol would be interesting future work.

Many of the complications that arose in the design of COGI and the controllers which implement it (CMC,CCC,CC) would be eliminated or mitigated by partitioning the single shared address space. If processors were allowed to access only local cluster memory and global memory, then all but one of the deadlock situations which arose and were solved in the COGI controller models would be eliminated. This modification would mean that cluster memory could not be shared between clusters. Since the HBSM architecture is designed for applications which have distinctly clusterable memory reference patterns, and since a large global memory would allow for intercluster sharing, this does not seem to be a debilitating restriction. An additional performance benefit which would result from the partitioning of memory is a decrease in global bus invalidation traffic. If CCC controllers were able to distinguish blocks which are cluster private from global blocks, then write notices to SharedUnmod cluster blocks would not require a global bus invalidation.

8.1.2 System requirements

The memory subsystem of a multiprocessor is only one part of a complex, interrelated system of hardware and software. Future studies of the hierarchical bus multiprocessor could involve other fundamental components of the system such as I/O, bus interconnection, processor design, hardware synchronization support, operating system, programming tools, and application programs.

Bus design and I/O are closely related to memory subsystem. To be economical an implementation of the HBSM multiprocessor would most likely use one of the established bus standards, such as VME, Multibus, or Futurebus. Support for multiple processors and cache coherence makes Futurebus a natural choice for the HBSM multiprocessor. The placement of I/O peripherals such as mass storage and network connections will be determined by the intended application of the multiprocessor. Disk storage would most likely be needed on each cluster, since the global bus would

be incapable of supporting the disk I/O requirements of multiple clusters. Network connections could either be distributed or centralized on the global bus, depending on whether the machine was intended for use in a multiple user time-sharing mode or in a single-user scientific mode.

Hardware support of synchronization primitives seems necessary for efficient operation of scientific applications. A common method of providing hardware synchronization is to provide atomic instructions such as Test&Set. The atomicity of the instruction is often guaranteed by locking the bus for the duration of the instruction. With a hierarchical bus architecture it may be advisable to limit synchronization operations to local cluster or global memory blocks. The possibility of integrating synchronization primitives with the cache coherence protocol should be investigated.

The design of the operating system for a hierarchical bus multiprocessor would be vitally important to the success of the machine. The issue of distributing the operating system kernel versus centralizing it would be the first major operating system design decision. Centralized kernels are easier to program, but can become a bottleneck; distributing the kernel on each processor would most likely be required. Task allocation could be done by maintaining a single task queue on each cluster. Processors would then take tasks from this queue when they were idle. Task migration represents a significant difficulty in clustered machines, due to the (assumed) affinity of a task for local memory.

Necessary support for programming would include intelligent compilers and linker/loaders, parallel symbolic debuggers, and performance evaluation tools. Knowledge of the clustered nature of the architecture would be important to the compiler to make decisions about the placement of variables in a processes virtual address space. The virtual memory management of the operating system would have to include the ability to map a tasks address space (particularly the code segments) to the local cluster. Debuggers and performance evaluation tools for parallel processing computers are currently active research problems. Lack of a global clock and a single thread of control make parallel debuggers difficult to create.

8.2 High Level Timed Petri Net Simulator

In order to debug the COGI cache coherence protocol and to gain some confidence in its correctness, detailed trace-driven simulation was performed with High Level

Timed Petri net models. This general purpose modeling tool can be used in many areas, but is particularly adept at representing two important characteristics of parallel computer models, concurrency and synchronization. The HLTPN simulator trades analytical power for modeling expressiveness so that complex systems can be more readily described. A current limitation to the Petri net simulator is the lack of a graphical user interface. Such an interface, added over the existing model editor, would considerably ease the creation of HLTPN models. If the graphical interface were to support the run-time features of the simulator, the iterative process of design/simulate/debug would be made faster and more direct.

8.3 Simulation Methodology

Verification of the probabilistic model with a more accurate trace-driven model indicates some deficiencies in the probabilistic model. Non-uniform memory access multiprocessors should be carefully modeled due to potential differences in cache behavior of different groups of memory. Extending the probabilistic model in Chapter 7 by separately characterizing cache performance of local and global blocks resulted in much better agreement between the two models. Unfortunately, the use of detailed trace-driven models will likely remain limited by storage space and simulation time, so it is important to know the strengths and weaknesses of simpler, faster models.

The size and detail of the trace-driven SILO model classifies it as a medium scale event scheduled simulation model. Further enhancement to this model would be difficult and would best be done in a process interaction paradigm. The size and complexity of the probabilistic SILO model leaves some room for extending it. For example, separately characterizing the cache behavior of local and global blocks increases the accuracy of the probabilistic model.

The tracing method developed to create architecture independent multiprocessor address traces proved adequate for trace-driven simulation. If the Tracer were extended to handle more general applications (currently, traced applications can only use barrier synchronization and static task allocation) the Tracer would certainly attract more interest. One possible strategy for this improvement is to introduce feedback from the multiprocessor simulator and to generate address traces in parallel with multiprocessor simulation. Feedback from the simulator could be used to direct the course of generally synchronized processes, and to make dynamic task allocation

possible. Tracing is currently limited to data address traces; instruction addresses are not traced. The Tracer could be readily extended to capture instruction traces by recording the program counter at every instruction. Since a delay value is currently incremented for each instruction, the facility for doing this is already in place.

Bibliography

Bibliography

- [1] S. Aggarwal and B. Gopinath. Foreword special issue on tools for computer communication systems. *IEEE Transactions on Software Engineering*, 14(3), Mar. 1988.
- [2] J. Archibald and J.-L. Baer. Cache coherence protocols: Evaluation using a multiprocessor simulation model. *ACM Transactions on Computer Systems*, 4(4):273–298, Nov. 1986.
- [3] J. K. Archibald. A cache coherence approach for large multiprocessor systems. In *Proceedings of the 1988 International Conference on Supercomputing*, pages 337–345, July 1988.
- [4] M. Azimi and C. Erickson. High level timed petri net simulator: A modeling and simulation tool for performance evaluation of concurrent systems. Technical Report MSU-ENGR-89-014, Michigan State University, Department of Electrical Engineering, Sept. 1989.
- [5] M. Azimi and C. Erickson. A software approach to multiprocessor address trace generation. In *Proceedings of the 14th International Computer Software and Applications Conference*, pages 99–105, Oct. 1990.
- [6] S. J. Baylor and B. D. Rathi. A study of memory reference behavior of engineering/scientific applications in parallel processing. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I:78–82, Aug. 1989.
- [7] G. Bell. Multis: A new class of multiprocessor computers. *Science*, 228, Apr. 1985.
- [8] R. R. Billig, S. S. Corbin, and R. L. Moore. A fast backplane cluster heralds a 1000-mips computer. *Electronic Design*, pages 81–86, July 1987.
- [9] J. Billington, G. Wheeler, and M. Wilbur-Ham. Protean: A high-level petri net tool for the specification and verification of communication protocols. *IEEE Transactions on Software Engineering*, 14(3), Mar. 1988.
- [10] L. M. Censier and P. Feautrier. A new solution to coherence problems in multi-cache systems. *IEEE Transactions on Computers*, 27(12), Dec. 1978.

- [11] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *IEEE Computer*, pages 49–57, June 1990.
- [12] H. Cheong and A. V. Veidenbaum. Compiler-directed cache management in multiprocessors. *IEEE Computer*, pages 39–47, June 1990.
- [13] G. Chiola. A software package for the analysis of generalized stochastic petri net models. In *International Workshop on Timed Petri Nets*, 1985.
- [14] G. Chiola. *GreatSPN users' manual version 1.3*. Technical report, Computer Science Department, Università di Torino, Sept. 1987.
- [15] I. Computer. Supercomputer hardware. *IEEE Computer*, 22(11):63–68, Nov. 1989.
- [16] A. Cumani. Esp - a package for the evaluation of stochastic petri nets with phase-type distributed transition times. In *International Workshop on Timed Petri Nets*, Torino, Italy, 1985.
- [17] H. Davis, S. Goldschmidt, and J. Hennessy. Tango: A software simulation and tracing system. Processor Tracing Methodologies Workshop International Symposium on Computer Architecture, 1990.
- [18] A. L. DeCegama. *The Technology of Parallel Processing: Volume 1*. Prentice Hall, Englewood Cliffs, New Jersey, 1989.
- [19] M. Dubois, F. A. Briggs, I. Patil, and M. Balakrishnan. Trace-driven simulations of parallel and distributed algorithms in multiprocessors. In *Proceedings of the 1986 International Conference on Parallel Processing*, pages 909 – 916, St. Charles, IL, 1986.
- [20] J. Dugan, G. Ciardo, A. Bobbio, and K. Trivedi. The design of a unified package for the solution of stochastic petri net models. In *International Workshop on Timed Petri Nets*, Torino, Italy, 1985.
- [21] S. J. Eggers and R. H. Katz. A characterization of sharing in parallel programs and its application to coherency protocol evaluation. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*. Computer Architecture News, May 1988.
- [22] S. J. Eggers and R. H. Katz. The effect of sharing on the cache and bus performance of parallel programs. In *Proceedings of the 3rd International Conference on Architectural Support for Programming Languages and Operating Systems*, pages 257–270, 1989.
- [23] J. Evans. *Structures of Discrete Event Simulation*. Ellis Horwood Limited, 1988.

- [24] R. S. Francis and I. D. Mathieson. A benchmark parallel sort for shared memory multiprocessors. *IEEE Computer*, 37(12):1619 – 1626, Dec. 1988.
- [25] S. J. Frank. Tightly coupled multiprocessor system speeds memory-access times. *Electronics*, January 12 1984.
- [26] J. Genrich and K. Lautenbach. System modeling with high-level petri nets. In *Theoretical Computer Science*, volume 13, pages 109–136. North-Holland, 1981.
- [27] J. R. Goodman. Using cache memories to reduce processor-memory traffic. In *Proceedings of the 10th International Symposium on Computer Architecture*, Trondheim, Norway, June 1983.
- [28] J. R. Goodman and P. J. Woest. The wisconsin multicube: A new large-scale cache-coherent multiprocessor. In *Proceedings of the 15th International Symposium on Computer Architecture*, pages 422–431, 1988.
- [29] S. Haridi and E. Hagersten. The cache coherence protocol of the data diffusion machine. In *Proceedings of PARLE*. Springer-Verlag, 1989.
- [30] J. L. Hennessy and D. A. Patterson. *Computer Architecture A Quantitative Approach*. Morgan Kaufman Publishers, Inc., San Mateo, California, 1990.
- [31] M. Holliday and M. Vernon. A generalized timed petri net model for performance analysis. *IEEE Transactions on Software Engineering*, 13(12), Dec. 1987.
- [32] K. Jensen. Coloured petri nets and the invariant-method. In *Theoretical Computer Science*, volume 14, pages 317–336. North-Holland, 1981.
- [33] K. Jensen. High-level petri nets. In *Applications and Theory of Petri Nets*. Springer-Verlag, Berlin, 1983.
- [34] M. Kumar and K. So. A study of memory reference behavior of engineering/scientific applications in parallel processing. In *Proceedings of the 1989 International Conference on Parallel Processing*, pages I:68–72, Aug. 1989.
- [35] E. D. Lazowska, J. Zahorjan, G. S. Graham, and K. C. Sevcik. *Quantitative System Performance: Computer System Analysis Using Queueing Network Models*. Prentice-Hall, Englewood Cliffs, New Jersey, 1984.
- [36] M. MacDougall. *Simulating Computer Systems*. The MIT Press, 1987.
- [37] S. Manoharan. *SILO: An event-based simulation platform*. University of Edinburgh, 1990.
- [38] M. Marsan, G. Balbo, A. Bobbio, G. Chiola, G. Conte, and A. Cumani. On petri nets with stochastic timing. In *International Workshop on Timed Petri Nets*, Torino, Italy, 1985.

- [39] E. M. McCreight. The dragon computer system: An early overview. In *NATO Advanced Study Institute on Microarchitecture of VLSI Computers*, pages 83–101. NATO Scientific Affairs Division, Urbino, Italy, 1985.
- [40] Meta Software Corporation. Design/cpn: A tool package supporting the use of colored petri nets. Internal publication, 1988.
- [41] M. K. Molloy. Performance analysis using stochastic petri nets. *IEEE Transactions on Computers*, 31(9), 1982.
- [42] J. Peterson. *Petri Net Theory and the Modeling of Systems*. Prentice-Hall, Englewood Cliffs, NJ, 1981.
- [43] A. M. Reiser and A. S. S. Lavenberg. Mean value analysis of closed multichain queueing networks. *Journal of the ACM*, pages 313–322, Apr. 1980.
- [44] W. Reisig. Petri nets with individual tokens. In *Applications and Theory of Petri Nets*. Springer-Verlag, Berlin, 1983.
- [45] C. H. Sauer and K. M. Chandy. *Computer Systems Performance Modeling*. Prentice Hall, Englewood Cliffs, New Jersey, 1981.
- [46] C. Scheurich and M. Dubois. Correct memory operation of cache-based multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 234–243, June 1987.
- [47] R. L. Sites and A. Agarwal. Multiprocessor cache analysis using atom. In *Proceedings of the 15th Annual International Symposium on Computer Architecture*, pages 186 – 195, May 1988.
- [48] K. So, F. Arema, D. A. George, and V. A. Norton. Psimul - a system for parallel simulation of parallel systems. In J. L. Martin, editor, *Performance Evaluation of Supercomputers*, pages 187 – 214. Elsevier Science Publishers B.V., 1988.
- [49] I. C. Society. *Futurebus+ Draft 8.2 P896.1 R/P8.2*. IEEE Computer Society Press, Feb. 1990.
- [50] P. Stenström. A cache consistency protocol for multiprocessors with multistage networks. In *Proceedings of the 16th International Symposium on Computer Architecture*, pages 407–415, 1989.
- [51] C. B. Stunkel and W. K. Fuchs. Traped: Producing traces for multicomputers via execution driven simulation. *Performance Evaluation Review*, 17(1):70 – 78, May 1989.
- [52] C. B. Stunkel, B. Janssens, and W. K. Fuchs. Address tracing for parallel machines. *IEEE Computer*, 24(1):31–38, Jan. 1991.

- [53] P. Sweazey and A. J. Smith. A class of compatible cache consistency protocols and their support by the ieee futurebus. In *Proceedings of the 13th International Symposium on Computer Architecture*, pages 414–423, June 1986.
- [54] C. K. Tang. Cache system design in the tightly coupled multiprocessor system. *Proceedings of the AFIP National Computer Conference*, 45, 1976.
- [55] P. J. Weinberger. Cheap dynamic instruction counting. *Bell System Technical Journal*, 63(8):1815 – 1826, Oct. 1984.
- [56] A. Wilson. Hierarchical cache / bus architecture for shared memory multiprocessors. In *Proceedings of the 14th International Symposium on Computer Architecture*, pages 244–252, June 1987.
- [57] A. Zenie. Colored stochastic petri nets. In *International Workshop on Timed Petri Nets*, Torino, Italy, 1985.