INFORMATION TO USERS

This manuscript has been reproduced from the microfilm master. UMI films the text directly from the original or copy submitted. Thus, some thesis and dissertation copies are in typewriter face, while others may be from any type of computer printer.

The quality of this reproduction is dependent upon the quality of the copy submitted. Broken or indistinct print, colored or poor quality illustrations and photographs, print bleedthrough, substandard margins, and improper alignment can adversely affect reproduction.

In the unlikely event that the author did not send UMI a complete manuscript and there are missing pages, these will be noted. Also, if unauthorized copyright material had to be removed, a note will indicate the deletion.

Oversize materials (e.g., maps, drawings, charts) are reproduced by sectioning the original, beginning at the upper left-hand corner and continuing from left to right in equal sections with small overlaps. Each original is also photographed in one exposure and is included in reduced form at the back of the book.

Photographs included in the original manuscript have been reproduced xerographically in this copy. Higher quality 6" x 9" black and white photographic prints are available for any photographs or illustrations appearing in this copy for an additional charge. Contact UMI directly to order.

UMI

A Bell & Howell Information Company
300 North Zeeb Road, Ann Arbor MI 48106-1346 USA
313/761-4700 800/521-0600

MEMORY LATENCY EVALUATION IN CLUSTER-BASED CACHE-COHERENT MULTIPROCESSOR SYSTEMS WITH DIFFERENT INTERCONNECTION TOPOLOGIES

By

Abu Sadath M. Asaduzzaman

A Thesis Submitted to the Faculty of

The College of Engineering

In Partial Fulfillment of the Requirements for the Degree of

Master of Science in Computer Engineering

Florida Atlantic University,

Boca Raton, FL, U.S.A.

August 1997

UMI
300 North Zeeb Road
Ann Arbor, MI 48103

Memory Latency Evaluation in Cluster-Based Cache-Coherent Multiprocessor Systems with different Interconnection Topologies

by

Abu Sadath M Asaduzzaman

This thesis is prepared under the direction of the candidate's thesis advisor, Dr. Imad Mahgoub, Department of Computer Science and Engineering, and has been approved by the members of his supervisory committee. It is submitted to the faculty of the College of Engineering and was accepted in partial fulfillment of the requirements for the degree of Master of Science in Computer Engineering.

SUPERVISORY COMMITTEE:

Imad Mahgoub
Thesis Advisor

M Yousuf

K Ganesan

A Ravishankar

A Ravishankar
Chairperson, Department of
Computer Science and Engineering

Dean, College of Engineering

John Sweeney
Dean, Graduate Studies and Research

7/18/97
Date

To: Shulakkhi Tahmina Zaman

ACKNOWLEDGMENTS

I express my profound gratitude and indebtedness to my academic and MS Thesis advisor Dr. Imad Mahgoub, Associate Professor, Department of Computer Science and Engineering, Florida Atlantic University who motivated me to do my MS Thesis. Dr. Mahgoub's constant guidance, valuable suggestions, advice and encouragement, kind help and strong support help me to finish this research work.

My special thanks and sincere gratitude to Dr. Mazin Yousif, IBM, who spent a lot of his time for me. For recent valuable information I always requested him and he responded very quickly, friendly, and helpfully. His whole-hearted help and unfailing interest throughout the progress of the work made me possible to carry out this study.

I also express my kind thanks and sincere gratitude to Dr. Mohammad Ilyas, Chairman and Professor, Department of Computer Science and Engineering, and Dr. K Ganesan, Associate Professor, Department of Computer Science and Engineering for their encouragement, and kind permission for working together.

Special thanks to Mr. Mahesh Neellakanta, UNIX system co-ordinator, Department of Computer Science and Engineering for providing computer facilities where almost all the computations were performed. Thanks to Dr. Sam Hsu, Associate Professor, Department of Computer Science and Engineering for allowing me to use Telecommunication Lab. I thankfully appreciate the cooperation of all staffs of Computer Science and Engineering Department and my friends, specially Shailender Bhandaram, Abu M Rahman, and James Kasper, who helped me designing the higher-level algorithm and collecting trace files.

Abu Sadath M Asaduzzaman

ABSTRACT

Author:	Abu Sadath M Asaduzzaman
Title:	Memory Latency Evaluation in Cluster-Based Cache-Coherent Multiprocessor Systems with different Interconnection Topologies
Institution:	Florida Atlantic University
Thesis Advisor:	Dr. Imad Mahgoub, Associate Professor
Degree:	Master of Science in Computer Engineering
Date:	1997

This research investigates memory latency of cluster-based cache-coherent multiprocessor systems with different interconnection topologies. We focus on a cluster-based architecture which is a variation of Stanford DASH architecture. The architecture, also, has some similarities with the STiNG architecture from Sequent Computer System Inc. In this architecture, a small number of processors and a portion of shared-memory are connected through a bus inside each cluster.

As the number of processors per cluster is small, snoopy protocol is used inside each cluster. Each processor has two levels of caches and for each cluster a separate directory is maintained. Clusters are connected using directory-based scheme through an interconnection network to make the system scaleable. Trace-driven simulation has been developed to evaluate the overall memory latency of this architecture using three different network topologies, namely ring, mesh, and hypercube. For each network topology, the overall memory latency has been evaluated running a representative set of SPLASH-2 applications. Simulation results show that, the cluster-based multiprocessor system with hypercube topology outperforms those with mesh and ring topologies.

TABLE OF CONTENTS

LIST OF FIGURES

LIST OF TABLES

Chapter 1

INTRODUCTION

In the early days of computer systems, most efforts in computer hardware were largely focused on the so-called traditional Von Neumann organization, which ran on stand-alone computers with single processors. It is obvious that the rapidly growing requirement for computing speed, system reliability, and cost-effectiveness will entail the development of alternative computers to replace the traditional uni-processors. Due to the availability of powerful microprocessors at low cost as well as significant advances in communication technology, multiprocessors computing, one of the latest dreams, is now possible. The overall performance of such a system heavily depends on the interconnection network and the application type.

Following general terms and concepts have been discussed - the advantages and the disadvantages of shared-memory multiprocessor system, different solutions to the cache coherence problem, and the role of different interconnection network topologies. Then the statement of this research work has been defined. Finally, the thesis organization has been presented.

1.1 Uni-processor versus Multiprocessor

Traditional uni-processor computers are unable to achieve the performance level required by large computing applications such as fusion modeling, weather forecasting, and aircraft simulation [1]. Shared-Memory multi-processors have emerged as an especially cost-effective way to provide increased computing power and speed. This kind is very popular and efficient due to some significant advantages.

1.1.1 Advantages of Shared-Memory Multiprocessors

Main advantages offered by shared-memory multiprocessors are [2]:

1) the simplest and the most general programming model,

2) use low-cost microprocessors economically inter-connected with shared-memory module,

3) all processors share the memory, various system resources such as I/O channels, control units, and code & data structures.

1.1.2 Disadvantages of Shared-Memory Multiprocessors

This system organization has three problems [2]:

1) Memory contention - when several request for the same memory address accrue at the same time,

2) Communication contention - when several processors (clusters) try to send information to other processors (clusters) using the network at the same time,

3) Long latency time - when multiprocessors with large number of processors tend to have complex inter-connection network. The memory latency time for such

networks (that is, the time required for a memory request to traverse the network) is long.

These problems contribute to increased memory access times and hence slow down the processors' execution speeds.

1.2 Locality of Memory References

Two main properties of the sequence of memory addresses generated by a program are (i) temporal locality and (ii) spatial locality. Temporal locality (or locality in time) means that memory addresses presently referenced by a program are likely to be referenced again in the near future. Spatial locality (locality in space) means that the addresses referenced by a program in a short period of time are likely to span a relatively small portion of the entire address space. The locality of memory references allows the cache to perform a vast majority of all memory requests (typically more than 95 percent), memory handles only a small fraction [2].

1.3 Cache Coherence Problem

Introduction of processor (or privet) caches helps greatly in reducing average latencies. Private data caches, which are small, fast memories physically located near a processor, exploit these memory-referencing properties to reduce the average time required to access the larger main memory. By temporarily storing a copy of a value from the main memory into the cache, which is being actively referenced by a program, caches amortize the time. The temporal locality and spatial locality allows the cache to perform a vast majority of all memory requests; memory handles only a small fraction.

Because of the sharing properties multiple copies of the same memory block can exist in different caches at the same time. To maintain a coherent view of the memory, these copies must be consistent. This is known as cache coherence problem (or the cache consistency problem) [2][3][4][5][6][7].

1.3.1 Solutions of Cache Coherence Problem

All known solutions to this problem can be classified into two main groups: (i) hardware-based and (ii) software-based. This traditional classification still holds, but solutions using combination of hardware and software become more frequent and promising. This thesis focuses the attention on the hardware-based solutions.

1.3.2 Software-based Schemes

These schemes mainly rely on the actions of the programmer, compiler, or operating system, in dealing with the coherence problem. The simplest but the most restrictive method is to declare non-cacheable blocks of shared data. More advanced methods allow the caching of shared data and accessing them only in critical sections, in a mutually exclusive way. Accesses to a shard variable by one processor may differ from those of other processors. The access may be one of the following types [2],

a) Read-only for any number of processors – cacheable by all processors

b) Read-only for any number of processors and Read-Write for exactly one processor - Write-through copy back, cacheable by the processor that has the write permission.

c) Read-Write for exactly one process – cacheable only by the Read-Write processor

d) Read-Write for any number of processors - non-cacheable

If considerable hardware support is provided, the software solutions are usually less expensive than their hardware counterparts. Some disadvantages are evident, specially in static schemes, where inevitable inefficiencies are incurred since the compiler analysis is unable to predict the flow of program execution accurately and conservative assumptions have to be made. Software-based solutions will not be discussed any further in this thesis.

1.3.3 Hardware-based Schemes

Although these schemes require an increased hardware complexity, their cost is well justified by significant advantages such as [7]:

(i) deal with coherence problem by dynamic recognition of inconsistency conditions for shared data entirely at the run-time. They promise better performance, specially, for higher levels of data sharing.

(ii) free the programmer and compiler from any responsibility about coherence maintenance, and impose no restrictions on any layer of software as they are totally transparent to software.

(iii) efficiently support the full range from small to large scale multiprocessors.

(iv) technology advances made their cost quite acceptable, compared to the system cost.

Some popular hardware-based schemes are:

(a) Directory protocol: Directory is maintained to store information of each block of main memory. We can directly access the memory block, if possible, no need of broadcast to all caches.

(b) Snoopy protocol: Consistency commands from one cache are broadcast to all other caches, and all other caches snoop the consistency commands.

(c) Coherence in Multilevel caches: Single level caches are unable to successfully fulfill two usual requirements - (a) to be fast and (b) large enough. Multilevel cache scheme is an unavoidable solution to the problem. Lower level caches are smaller but faster - their task is to reduce miss latency. Upper level caches are slower but much larger, in order to attain higher hit ratio and reduce the traffic on the interconnection network.

(d) Cluster-Based Cache-Coherence Protocol: For large shared-memory multi-processors, a number of processors and a small portion of memory are grouped together --- called cluster (processing node). Each processor may have multilevel caches; inside a cluster processors and the portion of memory are interconnected by a bus using snoopy scheme and clusters are connected through an interconnection network such as Ring and Mesh; directory is maintained using pointers to the clusters currently caching each memory block.

1.4 Cluster-based Protocols for Shared-Memory Systems

So far as we know, snoopy protocol has limited scalability and interference with the processor-cache write-path occurs. The main advantage of directory protocol is a broadcast is not required to find shared copies, but for a small-scale multiprocessors it is not efficient. In cluster-based cache-coherence protocol, we are taking the advantages of both snoopy protocol and directory protocol --- as a result this protocol has increased scalability and overall better performance.

Computationally the simulation of high-performance multiprocessors is expensive. The unit of simulated processor execution time requires many units of simulator time. For a parallel computer, simulator time increases in proportion to the total amount of work done in the simulated parallel machine. Moreover, simulators may incur substantial

overhead in scheduling and dispatching the large number of concurrent activities within a parallel machine. The overhead problem becomes specially bad when system components potentially interact on every memory address operation.

1.5 Cache-Coherent Interconnection Networks

In shared-memory multiprocessor systems, the real success mostly depends on the design of the network architecture that interconnects a large number of processors in an economical way. It is shown that a common bus does not suit hundreds of processors. Multistage networks have problems, because of the hardware complexity for many processors. So, a new network architecture is proposed, which is a combination of different networks to reduce network traffic. The Stanford DASH Multiprocessor used a combination of snoopy-directory scheme, where the processors in a node are connected by a bus and nodes are connected by mesh network.

1.6 Overall Memory Latency

Overall memory latency is the time required by the multiprocessor systems to finish a job. In this work, five representative SPLASH-2 applications are used to run the simulation program. Total time required to run one trace file is considered for each application. The interconnection network used to connect clusters is one of the important factors that affect the overall memory latency. The overall memory latency is investigated by using different interconnection network topologies.

1.7 Statement of the Problem

The performance quality required by large computing applications such as fusion modeling, and aircraft simulation [1] may not be achieved by any traditional uni-processor computer. Shared-memory multiprocessors have emerged as an especially cost-effective manner to provide increased computing power and speed.

To reduce the memory latency in a multiprocessor system, processor caches are introduced. These private data caches, which are small, fast memories physically located near a processor, exploit memory-referencing properties to reduce the average time required to access the larger main memory. By temporarily storing a copy of a block from the main memory, which is being actively referenced by a program, into a cache, the system amortizes the time. Because of the sharing properties, multiple copies of the same memory block can exist in different caches at the same time. To maintain a coherent view of the memory, these copies must be consistent. This is known as cache coherence problem [2][3][4][5][6][7]. To overcome cache coherence problem and to make the multiprocessor system scalable, cluster-based cache-coherent protocol is developed. In a cluster-based system, a (small) number of processors and a part of main memory are grouped together in a cluster.

In a cluster-based cache-coherent multiprocessor system, processors inside each cluster may be connected to each other by means of a bus or a ring. Clusters are connected using an interconnection network whose topology may influence the overall performance of the multiprocessor system. If the proper cluster-interconnection network topology can be selected for a cluster-based cache-coherent multiprocessor, then it is possible to decrease the memory latency and increase the overall performance of the system. Also, the choices of cache sizes influence the memory latency and performance. Before the data pollution point, memory latency decreases with the increase of the cache sizes used.

In this thesis, overall memory latency is evaluated in cluster-based cache-coherent multiprocessor systems with different interconnection topologies and cache sizes.

1.8 Thesis Contribution

In this thesis, we investigate memory latency of cluster-based cache-coherent multiprocessor systems for ring, mesh, and hypercube network topologies. Trace-driven simulation has been developed to evaluate the overall memory latency. Five representative SPLASH-2 applications have been used. The contribution of this thesis can be summarized as follows:

1. The DASH project and other related papers have been surveyed.

2. Variations of the DASH architecture with ring, mesh, and hypercube topologies along with a variation of the DASH cache coherence protocol have been proposed.

3. Trace-driven simulation for the proposed system has been developed.

4. Five representative SPLASH-2 applications have been studied and used to generate traces to drive the simulation program.

5. Memory latencies for the proposed systems have been evaluated using the selected five SPLASH-2 applications. Also, memory latency for different cache sizes has been evaluated.

6. Simulation results have been analyzed. The results show that, the proposed system with hypercube topology performed better than that with mesh and ring topologies. Also, the results show that, increasing the cache sizes decreases the memory latency.

1.9 Thesis Organization

This thesis work is organized as follows: in chapter 2, different hardware-based solutions of cache-coherence are discussed, among which DASH project and Lilja approach show significant improvements. In chapter 3 a modified approach of the DASH and STiNG architecture for cache coherence protocol used in multiprocessor systems is introduced to investigate the overall memory latency for different interconnection network topologies. Chapter 4 contains the assumptions and analysis of this simulation algorithm. Results from this simulation are discussed in chapter 5. Finally, conclusions and future directions are presented in chapter 6.

Chapter 2

CACHE COHERENCE IN SHARED-MEMORY MULTIPROCESSOR SYSTEMS

Cache coherence protocol is the appropriate solution of the problem of maintaining data consistency in shared-memory multiprocessors. This protocol helps in improving performance and scalability of the systems. It offers good performance since they deal with the problem fully dynamically. Great variety of schemes has been proposed, not many of them were implemented. This chapter is a survey of cache coherence schemes (hardware-based protocols) in shared memory multiprocessors. Section 2.1 is a brief introduction to hardware-based solutions. Cache-coherence policies are discussed in section 2.2. Section 2.3 is the introduction about consistency commands, snoopy protocols, and directory schemes.. Different directory schemes with their advantages and disadvantages are explained in section 2.4. Section 2.5 is the survey of DASH project. Section 2.6 is a collection of different important issues and comparisons affecting cache coherence in large-scale shared-memory multiprocessors from Lilja approach. Finally, section 2.7 contains the conclusion about this survey work.

2.1 Hardware-based Protocols

Hardware-based solutions are highly convenient because of their transparency of software [4][5]. This protocol includes snoopy protocols, directory schemes, and cache-coherent network architectures. Different cache coherence policies are needed for these protocols. Hardware mechanisms detect inconsistency conditions and perform actions according to a hardware implementation protocol. Software-based solutions attempt to avoid the need for complex hardware mechanisms. This chapter focuses on hardware-based solutions.

2.2 Cache-Coherence Policies

Data is divided into a number of equally sized blocks. A block is considered as the unit of transfer between memory and caches. This protocol allows any arbitrary number of copies of a block to exist at the same time. To maintain consistency of multiple copies two policies are used: write-invalidate and write update [2].

Write-Invalidate Policy: Read requests are carried out locally if a copy of the block exists. When a processor updates a block all other copies are invalidated. This policy works depending on the architecture used [e.g., Fig. 2.1b]. However, a subsequent update by the same processor can be performed locally in the cache, since copies no longer exist. Figure 2.1a shows four copies (one memory copy and three cached copies) of block x are presented in the system. Figure 2.1b shows processor 1 has updated an item in block x (the updated block is denoted by x') and other copies are invalidated (denoted by I). Now if any other processor issues a read request to an item in block x', then the cache attached to processor 1 supplies it because this is the only valid copy. Processor 1 can perform any number of update without issuing any invalidate request because there is no other valid copy in the system.

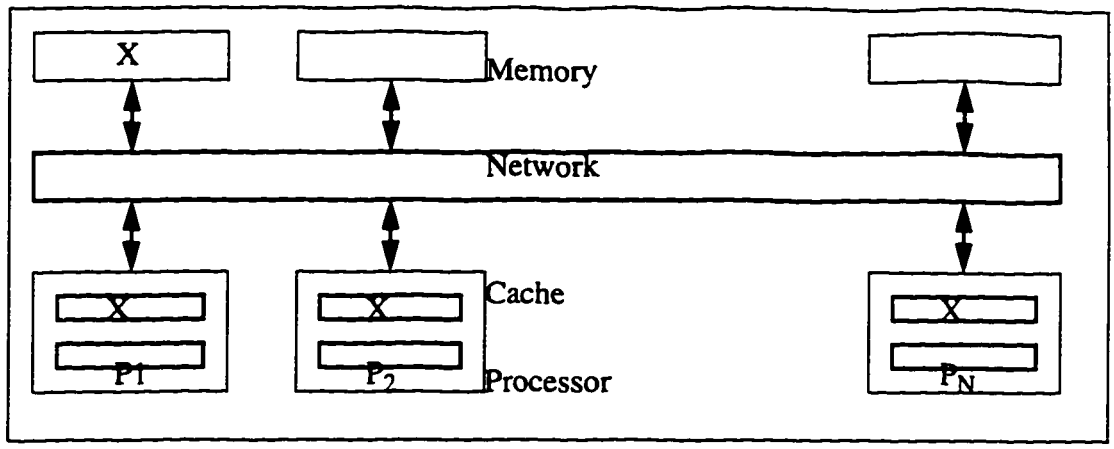


Fig. 2.1a: Consistent copies of a memory block
Memory and three caches store consistent copies of block X

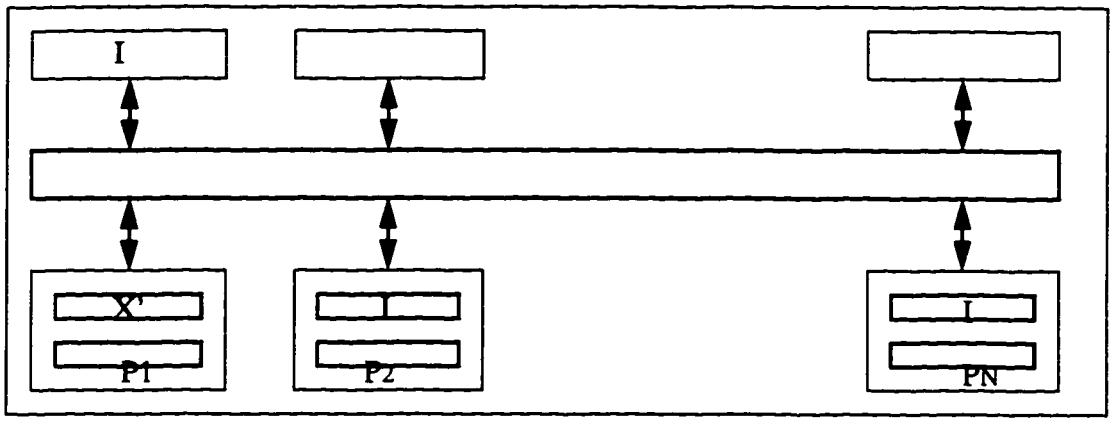


Fig. 2.1b: After Write-Invalidate is performed
All copies except processor 1's cached copy are invalidated (I), when processor 1 updates X (to X') if write-invalidate policy is used

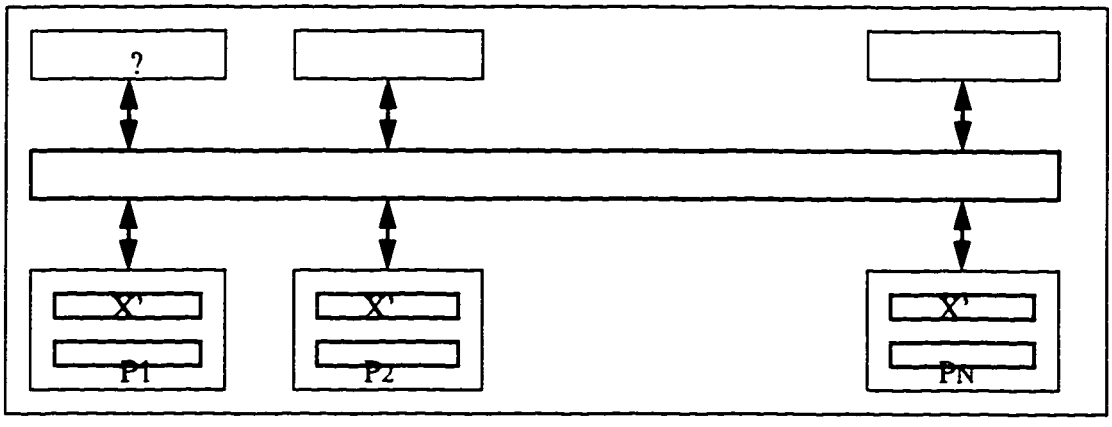


Fig. 2.1c: After Write-Update is performed
All copies (except the memory copy, which depends on the protocol) are updated if the write-update policy is used.

Write-Update Policy: When a processor updates a cached copy, it updates every other cached copies (if any). Whether the memory copy is updated or not depends on how this protocol is implemented. Figure 2.1c shows the new states of cached copies after updated by processor 1.

2.3 Snoopy Protocols

Cache invalidation and update commands are collectively referred to as consistency commands [2]. Write-Invalidate and Write-Update policies require that consistency commands be sent to at least those caches having copies of the block. In some networks (e.g. buses), it is feasible to broadcast consistency commands to all caches. These protocols are called snoopy cache protocols [2] because each cache snoops on the network for every incoming consistency command.

2.3.1 Write-Invalidate Snoopy Cache Protocols

The Write-once protocol, proposed by Goodman and reviewed by Archibald and Baer, is considered as the first Write-Invalidate snoopy cache protocol. The Illinois protocol, proposed by Papamarcos and Patel, and the Berkeley protocol, specifically designed for the SPUR Multiprocessor Workstation at the University of California at Berkeley, are other two Write-Invalidate protocol. In this section write-once protocol is discussed. Possible states for a cached copy used in write-once are [2]:

- Invalid - an inconsistent copy.
- Valid - any valid copy consistent with the memory copy.
- Reserved - data has been written exactly once and the copy is consistent with the memory copy, which is the only other valid copy.

• Dirty - data has been modified more than once and the copy is the only valid copy in the system.

Copy-back memory update policy was used in write-once protocol. This protocol used the following commands [2]:

• Read-Blk - Normal memory read block command, reads from a memory block.

• Write-Blk - Normal memory write block command, writes into a memory block.

• Read-Inv - Consistency command, reads a block and invalidates all other copies.

• Write-Inv - Consistency command, invalidates all other copies of a block.

Figure 2.2a explains different states of a cached copy and different commands used in write-once protocol.

The operation of write-once protocol can be explained by discussing the actions taken on processor reads and writes [2].

• Read hit - read hits always can be done locally in the cache and it does not result in state transitions.

• Read miss - (i) if a dirty copy exists, then the corresponding cache inhibits memory and sends a copy to the requesting cache. Both copies will change to valid and the memory is updated. (ii) if no dirty copy exists, then memory has a consistent copy and supplies a copy to the cache. This copy will be in the valid state.

• Write hit - (i) if the copy is in the dirty or reserved state, then the write can be performed locally in the cache and the new state will be dirty. (ii) if the state is valid, then a Write-Inv consistency command is broadcast to all caches, invalidate their copies. Write is performed when block is available and memory is updated. The resulting state is reserved.

• Write miss - the block either comes (i) from a cache with a dirty copy, which then updates memory, or (ii) from main memory. A Read-Inv consistency command is

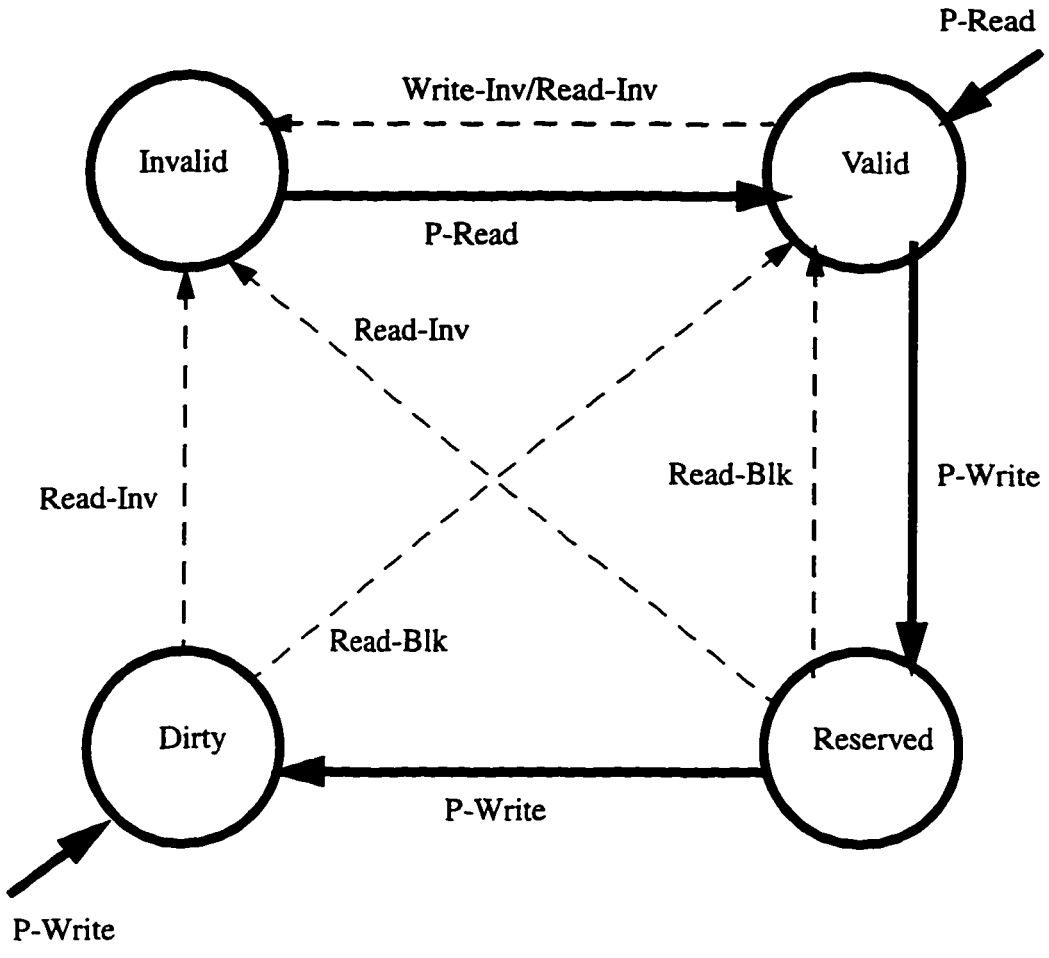


Fig. 2.2a: State-transition graph of cached copy for write-once protocol. Solid lines mark processor-initiated actions, and dashed lines mark consistency action initiated by other caches.

sent to get the block, which invalidates all cached copies (if any). The copy is then updated locally and the new state is dirty.

If the copy is dirty, then it has to be written back to main memory. Otherwise, no actions are taken.

2.3.2 Write-Update Snoopy Cache Protocols

The Firefly protocol, implemented in the Firefly Multiprocessor Workstation, is a good example of Write-Update snoopy cache protocol [2]. The Dragon protocol, proposed for the Dragon Multiprocessor Workstation from Xerox PARC, is another write-update protocol. Firefly protocol is discussed in this section. Possible states for the cached copy used in the firefly protocol:

- Valid-exclusive - this is the only cached copy that is consistent with the memory copy.
- Shared - this copy is consistent, and there are other consistent copies.
- Dirty - this is the only consistent copy, the memory copy is inconsistent.

A write-update consistency command updates all copies. This protocol uses copy-back update policy for private blocks and write-through for shared blocks. Figure 2.2b explains different states of a cached copy and different commands used in firefly protocol.

The operation of write-once protocol can be explained by discussing the actions taken on processor reads and writes.

- Read hit - read hits always can be done locally in the cache and it does not result in state transitions.
- Read miss - (i) if no cached copy exists, then the memory supplies the copy and the new state is valid-exclusive. (ii) if a dirty copy exists, then this cache supplies the copy because this is the only consistent copy, update the main memory. The new state is shared. (iii) if there is(are) shared copy(copies), then these caches

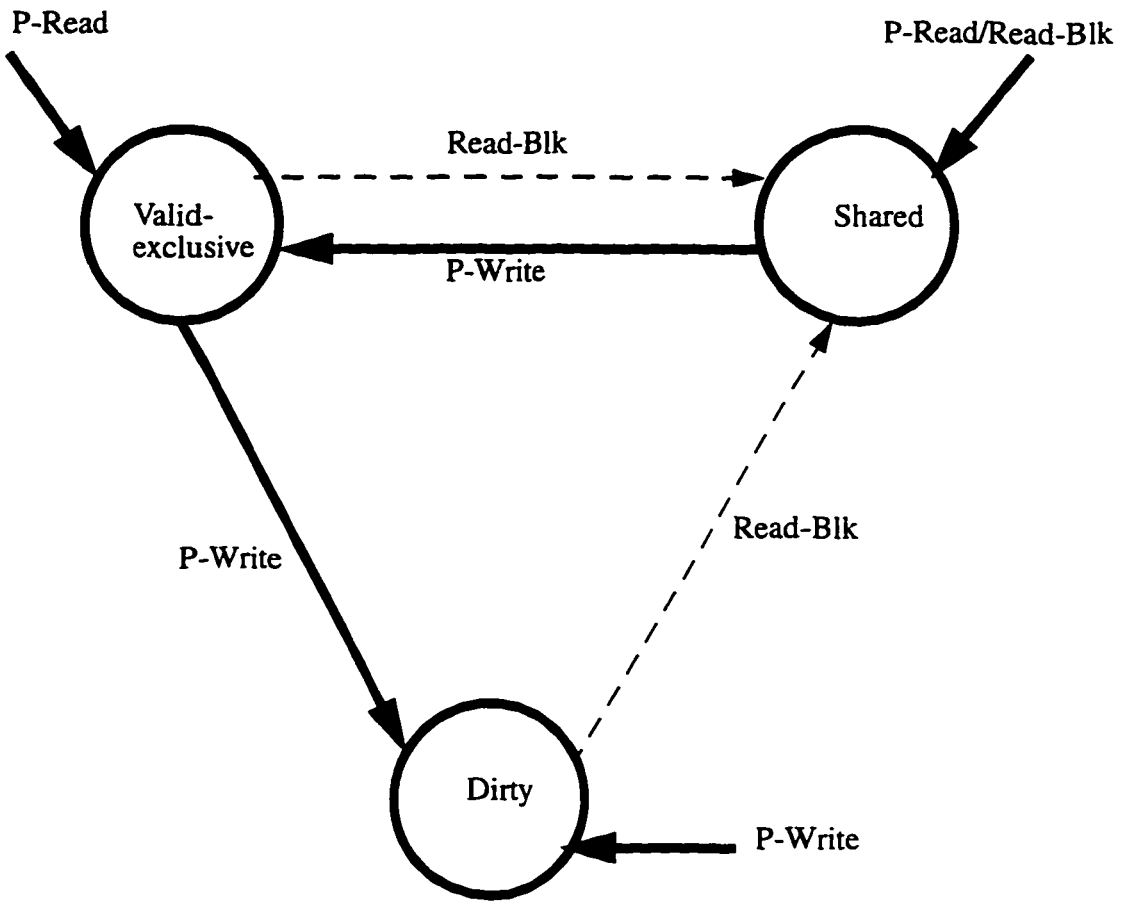


Fig. 2.2b: State-transition graph of cached copy for firefly protocol.
Solid lines mark processor-initiated actions, and dashed lines mark consistency actions initiated by other caches.

Supply the block by synchronizing the transmission on the bus. Resulting state, of course, remain unchanged.

• Write hit - (i) if the block is dirty or valid-exclusive, then the write is performed locally in the cache and resulting state is dirty. (ii) if the block is shared, all other copies (including the memory copy) are updated. If sharing is ceased, then the next state is valid-exclusive.

• Write miss - the required block is supplied either (i) from other caches or (ii) from main memory. If it comes from memory, then its loaded-in state is dirty. Otherwise, all other copies (including the memory copy) are updated and the new state is shared.

If the state is dirty, then the copy is written back to main memory. Otherwise, no actions are taken.

2.3.3 Implementation and Performance Issues

Because of simplicity and ease of implementation many commercial and bus-based multiprocessors have used snoopy cache protocols. The main differences between a snoopy cache and a uni-processor cache are (i) the cache controller, (ii)the information stored in the cache directory, and (iii) the bus controller. The miss ratio decreases until the block size reaches a certain point, the data population point, then it starts increase. Bus traffic per reference (in number of bus cycles, B) is proportional both to the miss ratio (M), and the number of words that must be transferred to serve a cache miss (L). Mathematically, B = K.M.L, where K is a constant [2]. If M decreases when L increases, then B will not necessary decrease. Simulation suggests using a small block size in bus-based snoopy protocol. For write-invalidate protocols, a cache miss can result from an invalidation initiated by another processor prior to the cache access - an invalidation miss. From Eggers and Katz's work it is clear that bus traffic in multiprocessors may increase

dramatically when the block size increases. For write-update protocols, the block size is not an issue because misses are not caused by consistency-related actions. The potential problem in this case is that write-update protocols tend to update copies even if they are not actively used. An important performance issue for write-invalidate policies concerns reducing the number of invalidation misses, and for write-update policies, an important issue concerns reducing the sharing of data to lessen bus traffic.

2.3.4 Limitations of Snoopy Cache Protocols

From Eggers and Katz's work it is proved that using large caches cannot entirely eliminate bus because of the consistency actions introduced as a result of data sharing. This sets an upper limit on the number of processors that a bus can accommodate. Snoopy cache protocols do not suit general interconnection networks, mainly because broadcasting reduces their performance to that of a bus.

2.4 Directory Schemes

To overcome the limitations of snoopy protocol and to improve the scalability of multiprocessor system, directory schemes are proposed. These systems multicast consistency commands exactly to the caches having a copy of the block. There is no need to broadcast consistency commands to those caches which do not contain a cached block. So a directory is required to track all copies of blocks and that is why these protocols are called directory schemes [2]. The main characteristic to distinguish directory schemes is that the global system-wide status information relevant for coherence maintenance is stored in some kind of directory. Upon the individual requests of the local cache controllers, centralized controller checks the directory, and issues necessary commands for

data transfer between memory and caches, or between caches themselves. Directory will keep status information up-to-date, so every local action affects the global state of the block must be reported to the central controller. The global directory can be organized in several ways: full-map directory, limited directory, and chained directory schemes [7].

2.4.1 Full-map Directory Schemes

The directory is stored in the main memory, and contains entries (presence bit) for each memory block. An entry points to exact locations of every cached copies of memory block, and keeps its status [7]. In this protocol, using the information from directory, coherence of data in private caches is maintained by sending directed messages to known locations, avoiding usually expensive broadcasts to every cache. Figure 2.3a shows a full-map directory organization, which indicates that both cache 1 and cache 2 hold a valid copy of memory block X, and other caches do not have this copy. The main advantage of full-map schemes are (i) that locating necessary cached copies is easy, and (ii) caches with valid copies are involved in coherence actions for a particular block. The main drawbacks of these schemes are (i) centralized controller is inflexible for system expansion by adding new processors, (ii) these schemes are not scaleable, and (iii) significant memory overhead when number of processors is large.

2.4.2 Limited Directory Schemes

In limited directory schemes the presence bit vector is replaced with a small number of identifiers pointing to cached copies [7]. Condition for saving memory space is $i*\log 2N < N$, where i is the number of pointer allowed and N is the total number of processors. For small i and large N, the size difference is significant. As shown in Figure 2.3b, at most two pointers are allowed (whereas in full-map at most N pointers are

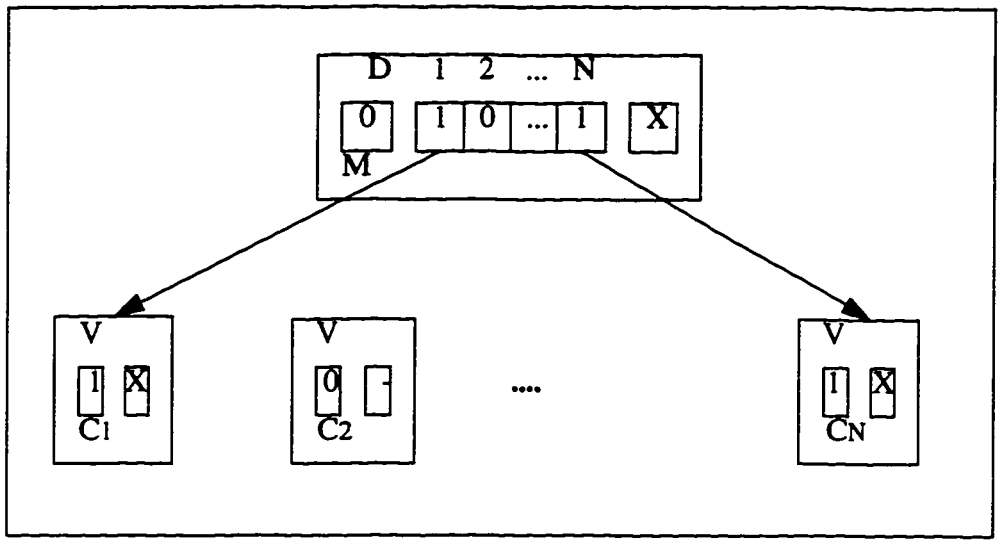


Fig. 2.3a: Full-map directory scheme

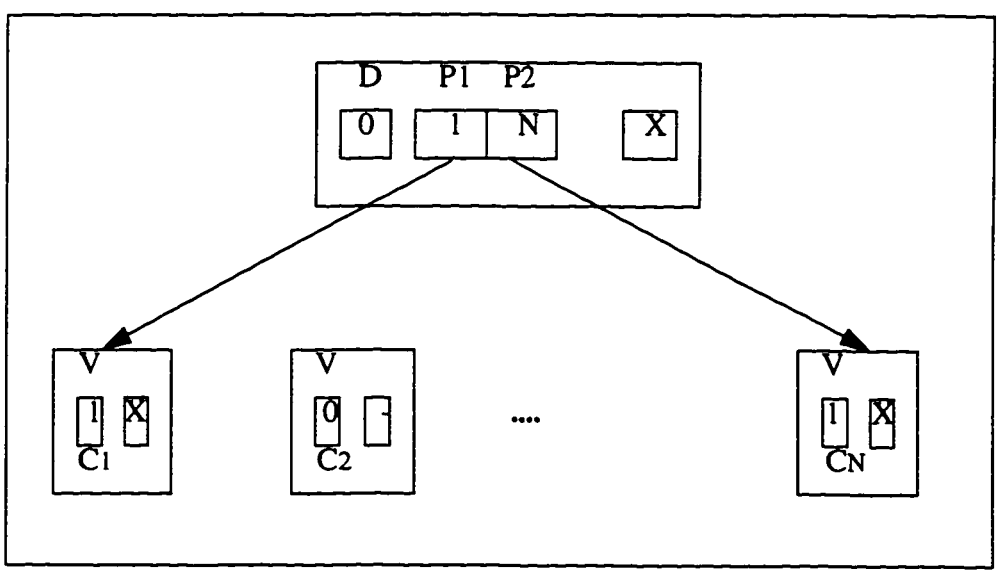


Fig. 2.3b: Limited directory scheme

allowed) , and only cache 0 and cache N hold valid copies. These schemes are also called partial-map schemes. Entries in limited directories contain a fixed number of pointers. When number of cached copies exceeds the number of pointers, special actions are required. Some schemes allow broadcasting to take care of this situation. If the scheme disallow broadcasts, one copy has to be invalidated, to free the pointer for a new cached copy. Memory overhead of this protocol is smaller when compared with full-map scheme, scalability is good; however, their performance heavily depends on sharing characteristics of parallel applications.

2.4.3 Chained Directory Schemes

Entries of chained directory schemes are organized in a form of (singly or doubly) linked lists, where all caches sharing the same block are chained through pointers into one list [7]. There is no limitation of number of cached copies. As shown in Figure 2.3c, chain directories are spread across the individual caches. Entry in main memory is used only to point to the head of the list and to keep the block status. Requests for the block are issued to the memory, and subsequent commands from the memory controller are usually forwarded through the list, using the pointer. The main advantage is, chained directory schemes are scaleable, while performance is almost as good as in full-map directory schemes.

2.5 Stanford DASH Project

The Computer Systems Laboratory at Stanford University has developed a cluster-based cache-coherence protocol for shared-memory multiprocessors called DASH

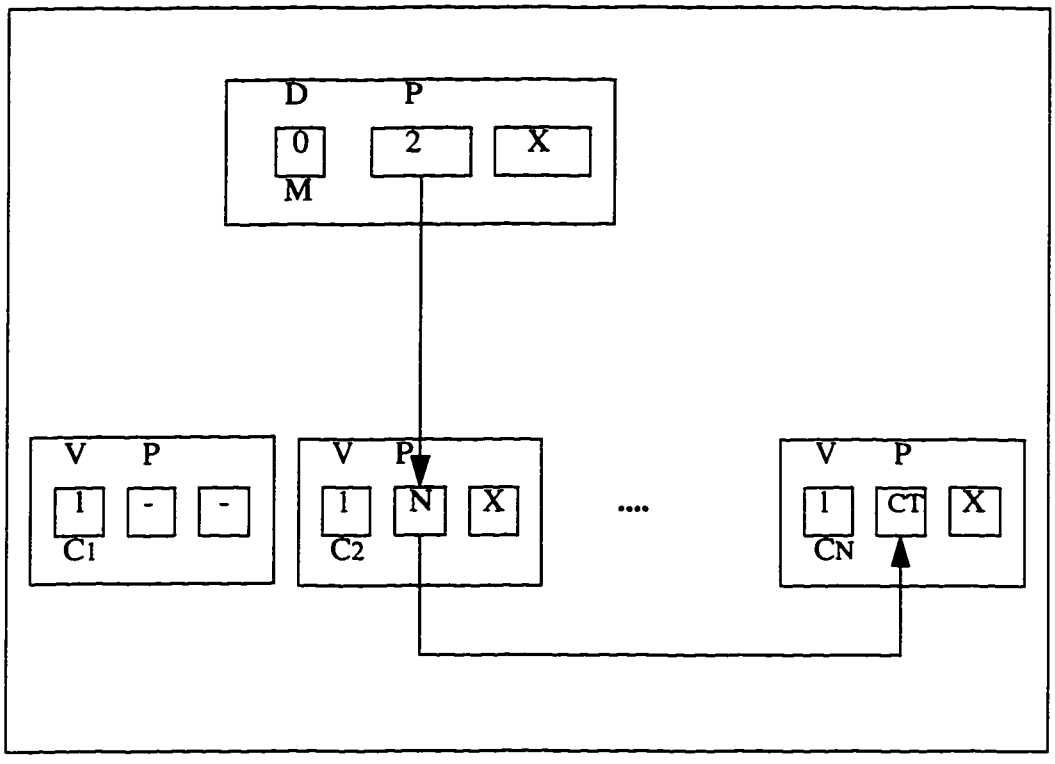


Fig. 2.3c: Chained directory scheme

Table 2.1: Notations used in Fig. 2.3a, 2.3b, & 2.3c

M - shared memory	V - valid bit
C - cache memory	D - dirty bit
X - data block	CT - chain terminator
P - pointer	N - # of processors

(Directory Architecture for SHared memory) [4][5]. The fundamental premise behind this architecture is that it is feasible to build large-scale shared-memory multiprocessors with hardware cache coherence. The DASH prototype system is the first operational machine to include a scaleable cache-coherence mechanism. This performance results from distributing the memory among processing nodes and using a network with scaleable bandwidth to connect the nodes. This architecture allows shared data to be cached, thereby significantly reducing the latency of memory access and yielding higher processor utilization and higher overall performance. A distributed directory-based protocol provides cache coherence without compromising scalability. The DASH prototype incorporates up to 64 high-performance RISC microprocessors to yield performance up to 1.6 billion instructions per second and 600 million scalar floating point operations per second.

David J. Lilja from Department of Electrical Engineering, University of Minnesots surveyed cache coherence mechanisms and identified several issues critical for design [6]. These design issues includes: (i) the coherence detection strategy, (ii) the coherence enforcement strategy, (iii) how the precision of block-sharing information can be changed to trade-off the implementation cost and performance of the memory system, and (iv) how the cache block size affects the performance of the memory system. Dr. Lilja used trace-driven simulations to compare the performance and implementation impact of these different issues.

2.5.1 The DASH Project Overview

The difference between the computing power of microprocessors and that of the largest supercomputers is decreasing and the price per performance advantage of microprocessors is increasing. This points to using microprocessors as the compute engines in a multiprocessor. But the problem is to build a machine that can scale up its

performance while maintaining the initial price per performance advantage of the individual processors. Scalability allows a parallel architecture to leverage commodity microprocessors and small-scale multiprocessors to build large scale machines. These large machines offer substantially higher performance.

High-performance processors are important to achieve both high total system performance and general applicability. DASH project used a parallel architecture to provide scalability to support hundreds to thousands of processors, high-performance individual processors, and a single shared address space [4][5]. A single address space enhances the programmability of a parallel machine by reducing the problems of data partitioning and dynamic load distribution. Caching of the memory, including shared write able data, allows multiprocessors with a single address space to achieve high performance through reduced memory latency.

Caching shared data introduces the cache-coherence problem. Although hardware support for cache-coherence has its costs, it also offers many benefits. Without hardware support, the responsibility for coherence falls to the user or the compiler. User may not like a complex programming model to handle caching. Handling the coherence problem in the compiler may be attractive, but currently cannot be done in a way that is competitive with hardware. The main problem with existing cache-coherent shared-address machines is the inability to scale effectively beyond a few high-performance processors.

The DASH project used cluster-based highly parallel architectures and directory-based coherence mechanism. They run a variety of applications efficiently and proved that DASH architecture permits single-address-space machines to scale and at the same time provides a flexible and general programming model.

2.5.2 The DASH Architecture

Directory-Based Cache-Coherence Scheme was first proposed in the late 1970s [4][5]. The original directory structures were not scaleable because in this scheme a centralized directory was used. This centralized directory quickly becomes a bottleneck. To overcome this drawback, DASH architecture partitioned and distributed the directory and main memory and used a new coherence protocol that can suitably exploit distributed directories. Moreover, DASH provides several other mechanisms to reduce and hide the memory operations.

The DASH architecture has a two-level structure as shown in Figure 2.4. At the high-level, the architecture consists of a set of processing nodes (clusters) connected through a mesh interconnection network; at the low-level, each cluster consists of a small number of high-performance processors and a portion of the shared memory interconnected by a bus. Intra-cluster cache coherence is implemented using a snoopy bus-based protocol, while inter-cluster coherence is maintained through a distributed directory-based protocol.

The cluster functions as a high-performance processing node. A bus-based cache protocol is chosen for implementing small-scale shared-memory multiprocessors because the bus bandwidth is sufficient to support a small number of processors. The grouping of multiple processors on a bus within each cluster amortizes the cost of the directory logic and network interface among a number of processors. At the same time, this grouping reduces the directory memory requirements by keeping track of cached blocks at a cluster as opposed to processor level. The directory-based protocol implements as invalidation-based coherence scheme. The directory keeps the summary information for each memory block, specifying its state and the clusters that are caching it. Except for the directory memory, DASH architecture is similar to many scaleable message-passing machines.

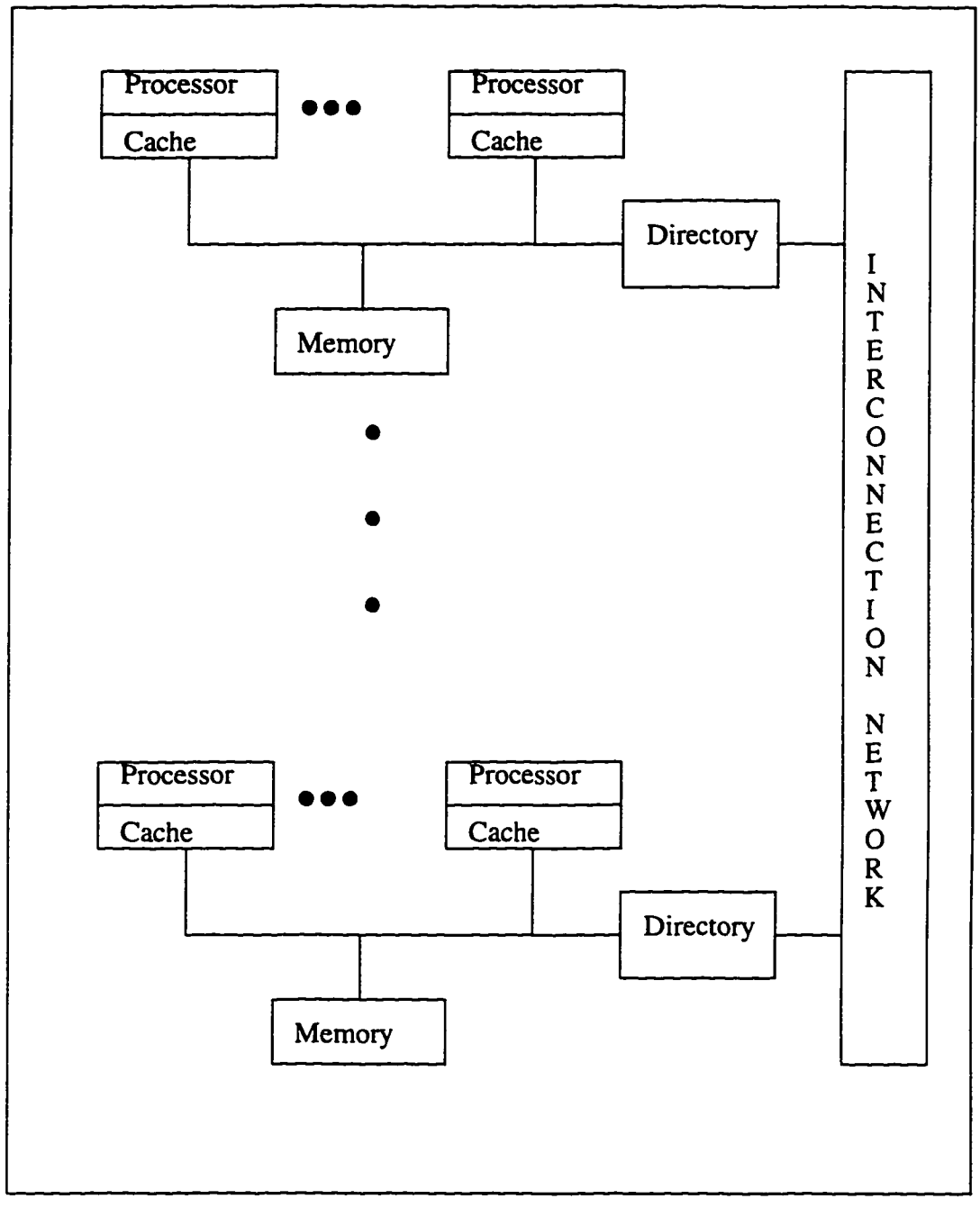


Figure 2.4: The DASH Architecture
A set of clusters are connected by a general interconnection network, directory memory contains pointers to the clusters currently caching each memory block.

2.5.3 The DASH Cache-Coherence Protocols

Normal Read and Write operations are considered in this protocol. When processor generates a request, current information is not enough to say whether it is a hit or miss operation.

Memory Hierarchy: A memory location may be in one of these three states [4]---

a) Un-cached - not cached by any cluster,

b) Shared - in an unmodified state in the caches of one or more clusters, or

c) Dirty - modified in a single cache of some cluster.

Logically DASH memory system is broken into four levels of hierarchy, (i) Processor Cache Level - Processor caches are designed to match the processor speed and support snooping from/to the bus. (ii) Local Cluster Level - This level includes some other processor's caches within the requesting processor's cluster. (iii) Home Cluster Level - The third level consists of the cluster that contains the directory and physical memory for a given memory address. For many accesses (e.g., private data references), the local and home cluster are the same. (iv) Remote Cluster Level - The fourth and final level for a memory block consists of the clusters marked by the directory as holding a copy of the block.

2.5.4 Memory Read Actions

If the requested block-address is present in the processor's cache, the cache simply supplies the data and the Read operation is done. If the Read request is not satisfied by the processor's cache, it is sent to the local cluster level. No state change occurs at the directory level.

If local cluster has at least one cached copy of that memory block, the request is satisfied within the cluster and no state change is required at the directory level. If the

request must be sent beyond the local cluster level, at first it goes to the home cluster corresponding to that address.

The home cluster examines the directory states of the memory location while simultaneously fetching the block from main memory. If the block is clean, the data is sent to the requester and the directory is updated to show sharing by the requester. If the location is dirty, the request is forwarded to the remote cluster indicated by the owner directory.

The dirty cluster replies with a shared copy of the data, which is sent directly to the request. Moreover a sharing Write-back message is sent to the home level to update main memory and change the directory state to indicate that the requesting and remote cluster now have shared copies of the data.

2.5.5 Memory Write Actions

If the block is in the writing processor's cache and the state of the block is dirty, the write can be performed immediately. Otherwise, a Read-exclusive request is issued on the local cluster's bus to obtain exclusive ownership of the line and retrieve the remaining portion of the cache line.

If one of the caches within the cluster already owns the cache line, then the Read-exclusive request is serviced at the local level by a cache-to-cache transfer. Processors within a cluster are allowed to alternately modify the same memory block without any inter-cluster interaction. If no local cache owns the block, then a Read-exclusive request is sent to the home cluster.

If the requested location is un-cached or shared the home cluster can immediately satisfy an ownership request. Also, if the requested location is in shared state, then all other cached copies must be invalidated. Invalidation requests are sent to those clusters

that have a copy of that block, at the same time, the home sends an exclusive data reply to the requesting cluster. If the directory indicates that the block is dirty, then the Read-exclusive request must be sent to dirty cluster.

If the required memory block is being shared, then the remote clusters receive an invalidation request to eliminate their shared copies; after receiving the invalidation, remote clusters send an acknowledgment to the requesting cluster. If the required block is dirty, the dirty cluster receives a Read-exclusive request. The remote cluster responds directly to the requesting cluster and sends a dirty-transfer message to the home indicating that the requesting cluster now holds the block exclusively.

When a writing cluster receives all the acknowledgments from the home or dirty cluster, it is obvious that all copies of the old data have been purged from the system. If the processor performs the write operation after receiving the acknowledgments, then the new value becomes available to all other processors at the same time. Note: invalidation is a round-trip method to multiple clusters which results potentially large delay. If we try to obtain higher processor utilization by allowing the processor to write immediately after ownership reply is received from the home, this may lead to inconsistencies with the memory model.

2.5.6 Maintaining Memory Consistency

Sequential consistency, which requires execution of the parallel program to appear as an interleaving of the execution of the parallel processes on a sequential machine. For many applications, such a model is too strict. The DASH prototype supports the release consistency model in hardware, which only requires the operations to have completed before a critical section is released. The main advantage of this scheme is it provides the user with a reasonable programming model; when the critical section is exited, all other

processors will have a consistent view of the modified data structure. Another important thing is it permits reads to bypass writes and the invalidation of different write operations to overlap, resulting in lower latencies for accesses and higher overall performance.

2.5.7 The DASH Implementation

A detailed software simulator of the system has been developed, but a hardware implementation is needed to understand this scheme.

DASH prototype cluster: a Silicon Graphics Power Station 4D/340 is used as the base cluster and the system consists of four Mips R3000 processors and R3010 floating-point coprocessors running at 33-MH [4]. Each R3000/R3010 combination can reach execution rates up to 25 VAX MIPS and 10 Mflops. Each CPU contains a 64-KB instruction cache and a 64-KB write-through data cache, which interfaces to a 256-KB second-level write-back cache. Both the first and the second-level caches are direct-mapped and support 16-B lines. The first level caches run synchronously to their associated 33-MH processors while the second level caches run synchronous to the 16-MH memory bus.

The second-level processor caches are responsible for bus snooping and maintaining coherence among the caches in the cluster. Coherence is maintained using am Illinois, or MESI protocol. 2X2 DASH system could scale to support hundreds of processors, but the prototype will be limited to a maximum configuration of 16 clusters.

DASH directory logic: The directory logic is used to implement the directory-based coherence protocol and to connect the clusters within the system. The directory logic is split between the two logic boards for outbound and inbound portions of inter-cluster transactions.

The directory controller (DC) board contains three major sections: (i) directory controller, which includes the directory memory associated with the cacheable main memory contained within the cluster, (ii) performance monitor, which can count and trace a variety of intra- and inter-cluster events, and (iii) the request and reply outbound network logic together with the X-dimension of the network itself. The directory information is combined with the type of bus operation, the address, and the result of snooping on the caches to determine what network messages and bus controls the DC will generate. The directory memory itself is implemented as a bit vector with one bit for each of the 16 clusters.

The reply controller (RC) board also contains three major sections: (i) reply controller, which tracks outstanding requests made by the local processors and receives and buffers replies from remote clusters using the remote access cache (RAC), (ii) pseudo-CPU, which buffers incoming requests and issues them to the bus, and (iii) inbound network logic and the Y-dimension of the mesh routing networks. RAC's primary role is the coordination of replies to inter-cluster transactions.

2.5.8 Interconnection Network

DASH coherence protocol does not rely on a particular interconnection network topology. For a scaleable architecture, the network itself must provide scaleable bandwidth and low latency communication. In DASH a pair of wormhole routed meshes is used as network, one handles request messages while the other is dedicated to replies. Wormhole routing allows a cluster to forward a message after receiving only the first flit (flow unit) of the packet, greatly reducing the latency through each node. An important constraint on the network is that it must deliver request and reply messages without deadlocking. DASH prototype cannot guarantee a deadlock-free mesh network because of

the limited buffering on the directory boards and the fact that a cluster may need to generate an outgoing message before it can consume an incoming message. DASH avoids these deadlocks through three mechanisms: (i) reply messages can always be consumed, (ii) the independent request and reply meshes eliminate request-reply deadlocks, and (iii) a back-off mechanism breaks potential deadlocks due to request-request dependencies.

Software support: For effective use of a large-scale multiprocessors a comprehensive software development is essential. DASH focused on: operating systems, compilers, programming languages, and performance debugging tools. DASH supports a full-function UNIX operating system. Developed in cooperation with Silicon Graphics, the DASH OS is a modified version of the existing operating system on the 4D/340 (Irix, a variation of UNIX System V.3) to accommodate the hierarchical nature of DASH, where processors, maim memory, and I/O devices are all partitioned across the cluster. They worked on several tools at the user level to aid the development of parallel programs for DASH. They also developed a parallel language called jade to find parallelism beyond the loop level, which allows a programmer to easily express dynamic coarse-grain parallelism. Using Jade can significantly reduce the time and effort required to develop a parallel version of a serial application. They also developed a suite of performance monitoring and analysis tools. The high-level tools can identify portions of code where the concurrency is smallest or where the most execution time is spent, also they provide information about synchronization bottlenecks and load balancing problems. The low-level tools will couple with the built-in hardware monitors in DASH.

2.5.9 The DASH Performance

The three applications simulated by DASH are Water, Mincut, and MP3D. Water is a molecular-dynamics code that computes the energy of a system of water molecules.

Mincut uses parallel simulated annealing to solve a graph-partitioning problem. MP3D models a wind tunnel in the upper atmosphere using a discrete particle-based simulation. Three steps of DASH performance: (i) the latency for memory accesses serviced by the three lower levels of memory hierarchy, (ii) speedup for three parallel applications running on a simulation of the prototype using one to 64 processors, and (iii) we present the actual speedups for these applications measured on the initial 16-processor DASH system.

While caches reduce the effective access time of memory, the latency of main memory determines the sensitivity of processor utilization to cache and cluster locality and indicates the costs of inter-processor communication. Applications for large-scale multiprocessors must utilize locality to realize good cache hit rates, minimize remote accesses, and achieve high processor utilization.

Water and Mincut achieved reasonable speedup through 64 processors. For Water, the reason is that the application exhibits good locality. For Mincut, good speedup results from very good cache hit rates. MP3D did not exhibit good speedup because, the encoding of the MP3D application requires frequent inter-processor communication, thus resulting in frequent cache misses. On average, about 4% of the instructions executed in MP3D generate s read miss for a shared data item.

DASH achieved reasonable speedup when going from 16 to 32 and 64 processors. Even on MP3D, caching is beneficial. They also used several other applications on their 16-processor prototype like two hierarchical n-body applications, a radiosity application from computer graphics, a standard-cell routing application from very large scale integration computer-aided design, and several matrix-oriented applications, including one performing sparse Cholesky factorization.

DASH group commented, "Our experience with the 16-processor machine has been very promising and indicates that many applications should be able to achieve over 40 times speedup on the 64-processor system."

2.6 Cache Coherence in Large Scale: Issues and Comparisons

The most important consideration, choosing a cache coherence scheme for a multiprocessor system, are performance (how effective it allows the caches to be in reducing the average delay), implementation cost (how much memory is required to store the cache block sharing information, and the complexity of the control logic [6]. Of course different schemes have significantly different trade-offs in cost and performance. Basic major factors, which have great impacts on the system performance:

(i) The coherence detection strategy: detects a possibly incoherent memory access either dynamically at run time or statically at compilation time.

(ii) The coherence enforcement strategy: updating or invalidating, which are used to ensure that stale cache entries are never referenced by a processor.

(iii) Precision of block-sharing information can be changed to trade-off the implementation cost and the performance.

(iv) Affects of cache block size on performance of the memory system.

2.6.1 Coherence Detection Strategy

The dynamic coherence detection strategies solve the coherence problem by examining the actual memory addresses generated by a program at run-time and dynamically keeping track of which processors have a copy of which blocks. The static coherence schemes try to predict which memory addresses may become stale by analyzing the program's referencing behavior when it is compiled [6].

2.6.2 Coherence Enforcement Strategy

The actual method used by the coherence scheme to ensure that no processor accesses a state-memory location. The simplest solution is to make all shared-writable memory locations non-cacheable so there will be no duplicate copy; this approach may significantly reduce performance [6]. Other strategies such as update and invalidate always allow shared write able memory locations to be cached, but either update or invalidate stale-cache entries before they are referenced again. In update schemes, the new value of the shared location is distributed to all processors. So, it reduces additional miss but increases network traffic. According to the invalidate scheme, all other cached copies are marked invalid when copy is updated. It reduces network traffic, but it introduces the extra delay of another miss if the block is reused.

2.6.3 Precision of Block-Sharing Information

Coherence mechanism that dynamically determines which memory references need coherence actions track the state and sharing characteristics of every memory block referenced by the program [6]. When a block needs to be invalidated, these exact mechanisms send invalidation messages only to those processors that actually have a cached copy of the block. Some recent tagged directories further reduce the directory memory requirements by maintaining sharing information only for blocks that are actually cached.

2.6.4 Cache Block Size

The cache block size or line size is the number of consecutive memory words updated or invalidated as a single unit. The fetch size is the number of words moved from

the main memory to the cache on a miss. When block size is less than the fetch size, increasing the size of block can reduce the miss ratio because of spatial locality. When the block size becomes too large (exceeds the data pollution point), the miss ratio increases since the probability of using the additional fetched data becomes smaller than the probability of reusing the data replaced [6]. The block size that minimizes the average memory delay generally is smaller than the block size that minimizes the miss ratio because the additional time required to transfer the large blocks can overwhelm the latency to receive the first word.

2.7 Summary

Following papers have been studied, "A Survey of Hardware Solutions for Maintenance of Cache coherence in Shared Memory Multiprocessors"[7], "A Survey of Cache Coherence Schemes for Multiprocessors" [2], and "Cache Coherence In Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons" [6]. This is just a comprehensive overview of hardware-based solutions to the cache coherence problem in shared memory multiprocessors. For more powerful and more efficient shared memory multiprocessors, the architects and designers should be very careful about the problems and the impact of the applied solution on system performance.

Overall time required to access the memory can be reduced significantly by adding private or processor caches. These private caches introduce cache coherence problem. Hardware-based solutions are discussed in this chapter. Different important architectural issues that affect the performance and implementation cost of a cache coherence schemes are surveyed. Trace-driven simulations have been used to quantify the performance impact of these different factors. The DASH project has demonstrated a coherence mechanism

that incorporates both a bus-based snooping protocol and directory-based coherence scheme, it gives the programmer a choice of both updating and invalidating coherence enforcement strategies.

Despite of the considerable advancement of the field, it still represents a very interesting research field. The RISC (Reduced Instruction Set Computer) microprocessors with an increased memory bandwidth requirement will put an increased burden on the memory system for future multiprocessors. So, multiprocessor caches are and will remain an important topic in the coming years. No doubt DASH project is a new direction in multiprocessor system. In the research paper a new approach is implemented which gives better performance.

Chapter 3

SYSTEM ORGANIZATION AND CACHE COHERENCE PROTOCOL

A great variety of schemes have been proposed to solve the well-known cache-coherence problem in shared memory multiprocessors. The main objective is to improve both the system performance and scalability. Hardware methods are highly convenient because of their transparency to software.

In this chapter, we investigate a clustered architecture that consists of a set of nodes connected by a general inter-connection network [10][11][12]. Inside each cluster, processors can submit their requests and/or receive requests from other processors through the bus that connects them. Clusters can exchange messages/data among themselves using the inter-connection network. A bus-based architecture is chosen in a cluster for performance and simplicity. Section 3.1 discusses different interconnection networks. A description of the proposed architecture is presented in Section 3.2. In

Section 3.3, we discussed the cache coherence protocol for this multiprocessor system, where actions required for memory read or memory write operation are explained. Finally, Section 3.4 presents the summary of this chapter.

3.1 Overview of Topologies Used

Processors in any multiprocessor system are connected by effective, efficient, and reliable communication network. Each processor can exchange messages and/or data with others through this network. There are many communication networks being proposed. Here, in this work, three of them are considered:

(i) Ring,

(ii) Mesh, and

(iii) Hypercube.

These three networks have been chosen because of their simplicity in design and reliability in operation. DASH project from Stanford University used mesh and STiNG from Sequent Computer Systems Inc. used ring network. Double connected networks have been used to increase the system reliability and to decrease routing deadlock.

The main focus of this work is to measure and compare the overall memory latency among three different network topologies. Each network is considered with its own characteristics, but other parameters such as channel speed, dedicated path between any two nodes, and so on are same for all three networks.

3.1.1 Ring

A collection of homogenous nodes interconnected via a set of high-speed point-to-point links. Each node or cluster contains processors, caches, memories, directories, and I/O busses. Every processor in every node has a common view of the system-wide memory and I/O address space. Within a node, cache coherence is maintained using MESI snooping protocol. Although there are dedicated paths between any two nodes, intermediate node(s) may be required to establish the path. So information from node 0 to node 3 in a 8-cluster system goes through link 0-1, node 1, link 1-2, node 2, and link 2-3. Link speed is considered as 1Gbyte per second and delay due to each intermediate node is considered as 30 neno seconds.

Some of the advantages, simple and reliable network, high data transmission rate, and low transmission error rate. Some disadvantages are, waste of bandwidth, and if not heavily utilized, many empty paths are unused.

3.1.2 Mesh

In this network, the clusters are arranged in a two dimensional (X-Y) fashion. Two adjacent clusters are connected through two communication channels. So, the number of unidirectional links, bandwidth, and cost is approximately 4*N for large N, where N is the total number of clusters. A linear growth in the cost as a function of size is usually

acceptable, since the processor cost is also linear with N. To send information from source to destination, at first the information is routed in X-direction, then in Y-direction.

Some advantages, for computation-intensive problems that map well onto a two-dimensional mesh, this network is attractive, and simple and reliable network. Some disadvantages are, in a NxN mesh if data have to go from one corner to the diagonally opposite corner, 2N hops are required, where the average will be about half of the worst case.

3.1.3 Hypercube

In an n-degree hypercube (also called an n-cube), 2^n nodes are arranged in an n-dimensional cube, where each node is connected to n other nodes. In this type of architecture, the clusters are the nodes of a hypercube and a hypercube edge corresponds to a bidirectional communication link between two clusters. Each of the 2n nodes of an n-cube are assigned a unique n-bit address ranging from 0 to 2n - 1 such that the address assigned to two adjacent nodes differ only in 1 bit position. The maximum distance between any two nodes in an n-cubes is n hops.

Some advantages of this network, fairly easy to build, and reduced worst-case communication delay. Some disadvantages, each time the dimension of the multiprocessor system increases by 1, number of processors becomes double; total number of processors is always a power of 2 (two).

3.2 The System Architecture

In snoopy scheme, as mentioned above, the information about which processor caches have cached copies are available within the clusters. So, snoopy schemes require that all caches see every memory request from every processor. This inherently limits the scaleability of these systems because the common bus structures due to its limited Bandwidth. With high-performance RISC processors this saturation can occur with just a few processors.

In directory scheme the need to broadcast every memory request to all processor caches has been removed. The directory maintains pointers to the processor caches holding a copy of each memory block. Only the caches with copies can be affected by an access to the memory block. Thus, the processor caches and interconnect will not saturate due to coherence requests. Moreover directory schemes are not dependent on any specific interconnection network.

Figure 3.1a shows the general overview of the higher level organization. This architecture consists of a number of processor nodes or clusters connected through directory controllers to a low-latency interconnection network. Each cluster consists of a small number of high-performance processors and a portion of the shared memory interconnected by a bus. It is essential to distribute the memory among the clusters to allow the system to exploit locality. Multiprocessing within the cluster can be viewed either as increasing the power of each cluster or as reducing the cost of the directory and network interface by amortizing it over a larger number of processors.

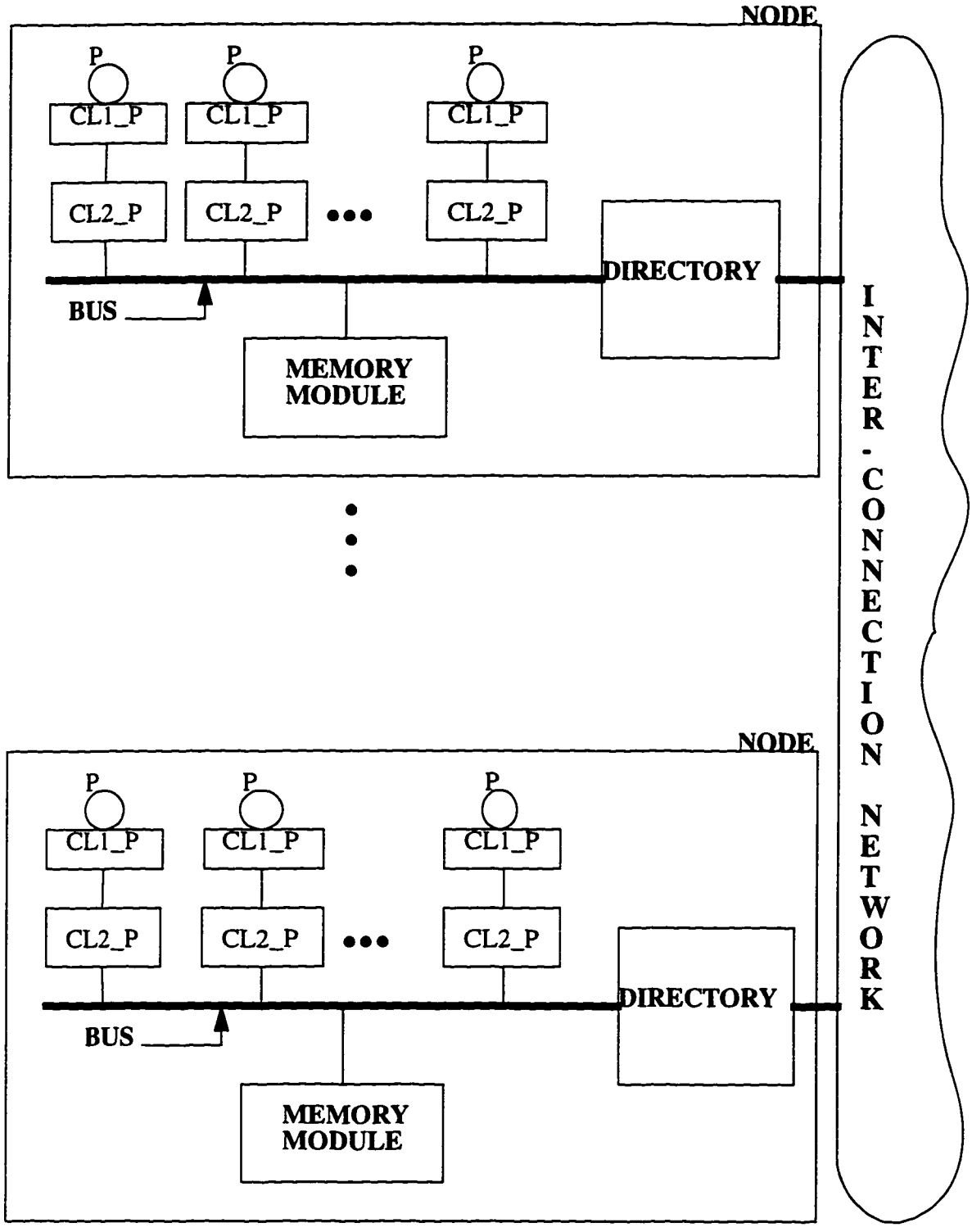


Figure 3.1a: Simulation architecture with a general interconnection network

In this simulation 16 processors are considered to form a multiprocessor system. Two cases are implemented:

case 1 - four processors are arranged to be in a cluster as shown in Figure 3.1b, resulting in a system with 4 clusters, and 4 processors in a cluster.

case 2 - two processors are arranged to be in a cluster as shown in Figure 3.1c, resulting in a system with 8 clusters, and 2 processors in a cluster.

3.2.1 Ring Network Topology

Figures 3.2a and 3.2b show cluster-based multiprocessor systems using a double connected ring network for 4 clusters and 8 clusters, respectively, within each cluster, a portion of memory and a directory is assigned. Each cluster is connected to both rings with two network interfaces, one for request and other one for acknowledgment. There are dedicated paths between any two nodes, as mentioned earlier. When one cluster needs to send any message/data to another cluster, it checks the link or path; when it gets the link free, it starts transmitting information regardless of the distance between source and destination. If the path is busy, it waits for the link to be free. When information passes through the link, every node checks the destination indicator, all intermediate nodes let the message go through to the next node. So, there is a delay associated with each intermediate node. The destination node makes a copy of the message, sends the

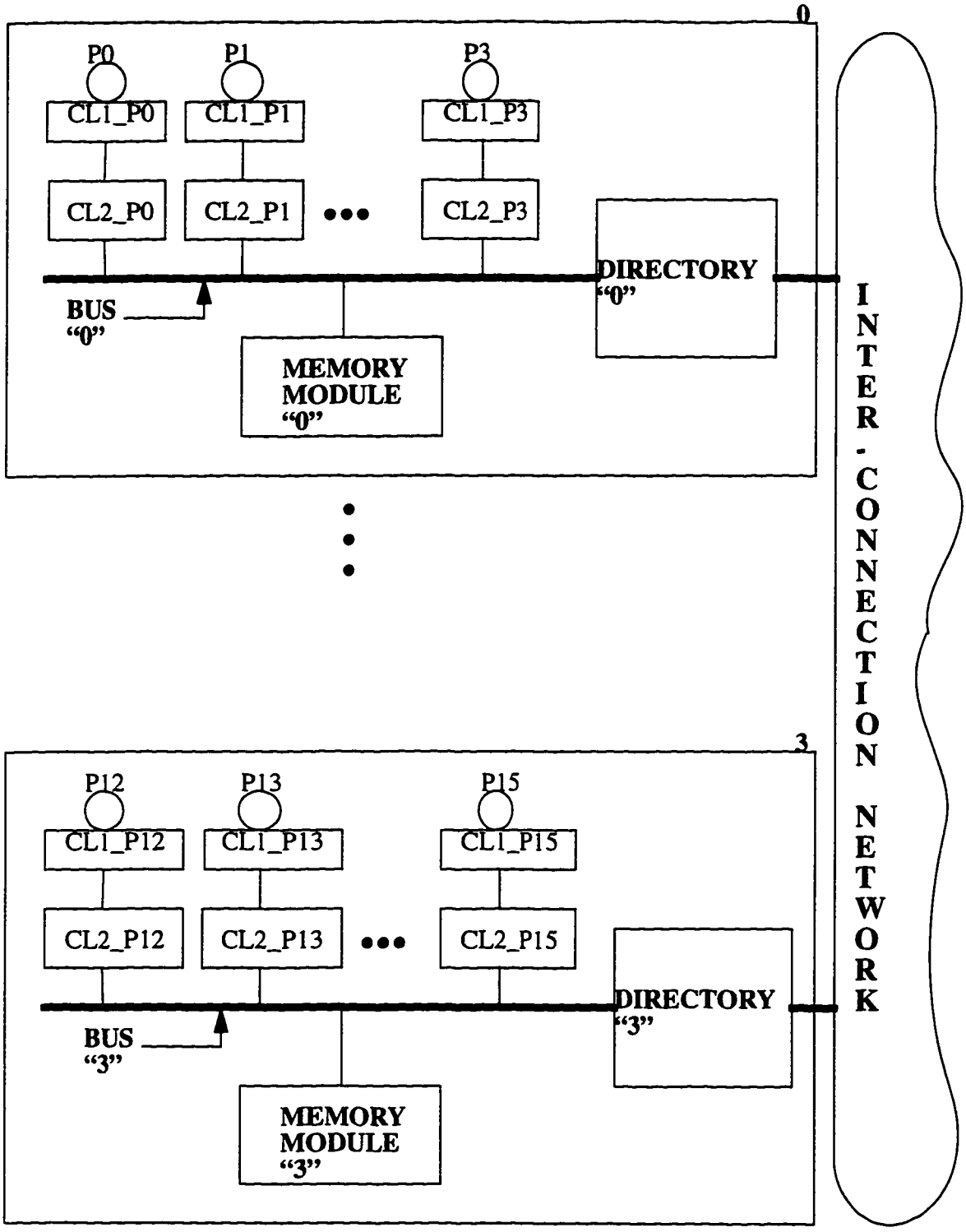


Figure 3.1b: Proposed architecture for 16 processors in 4 clusters

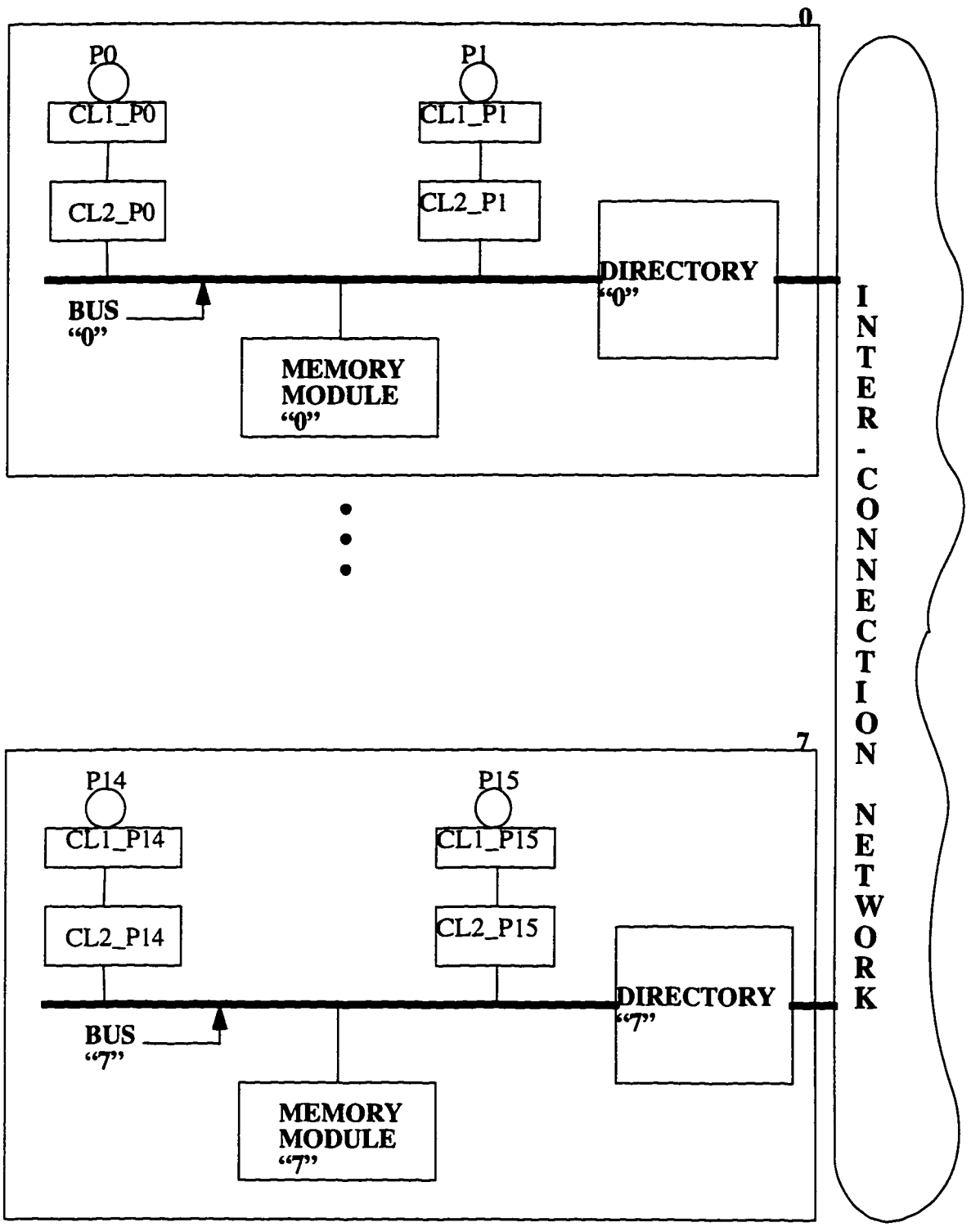


Figure 3.1c: Proposed architecture for 16 processors in 8 clusters

RING NETWORK:
Garb the first free token, irrespective of the traveled distance.

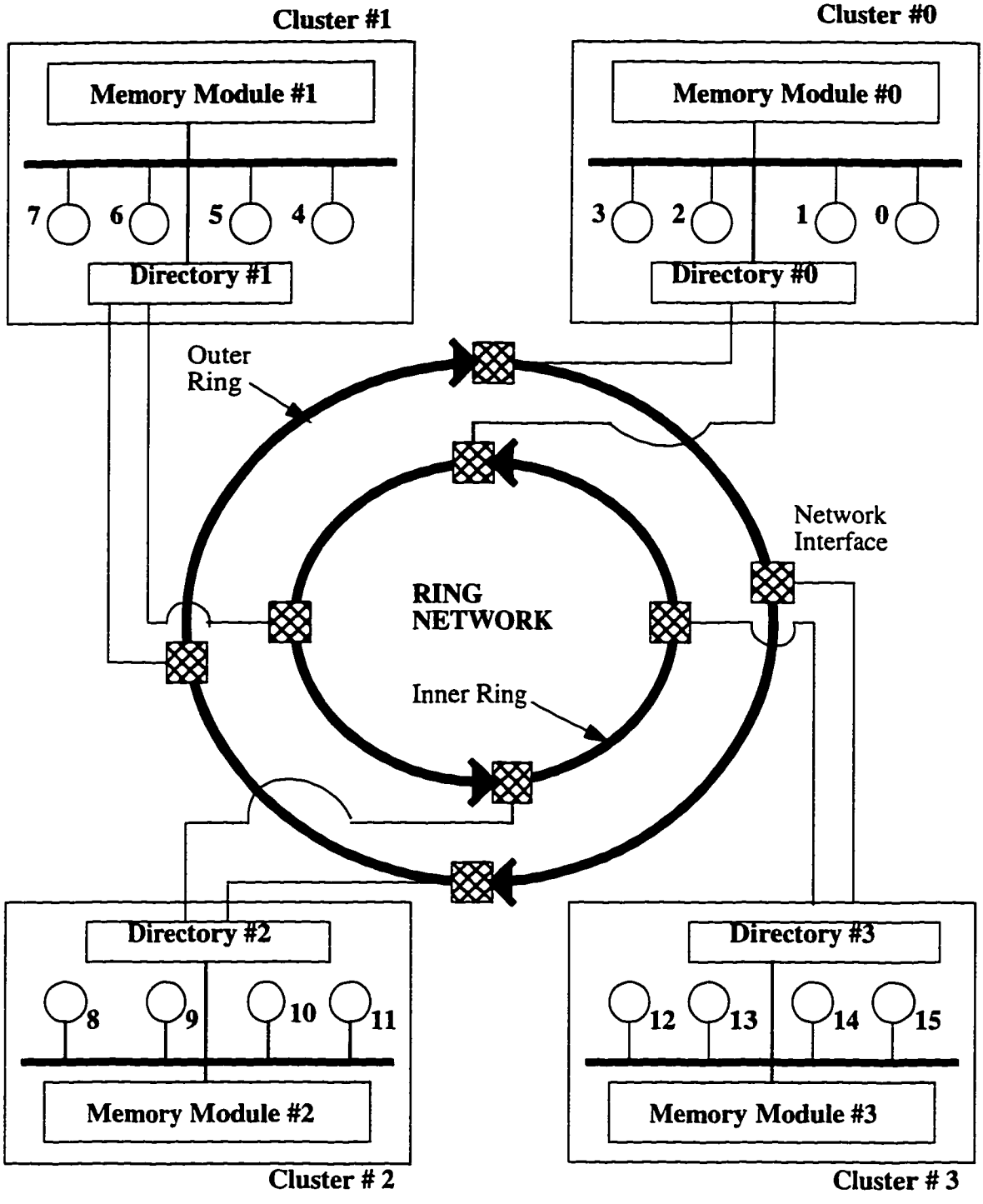


Figure 3.2a: Cluster-based multiprocessor (16 P, 4 C) system using ring network

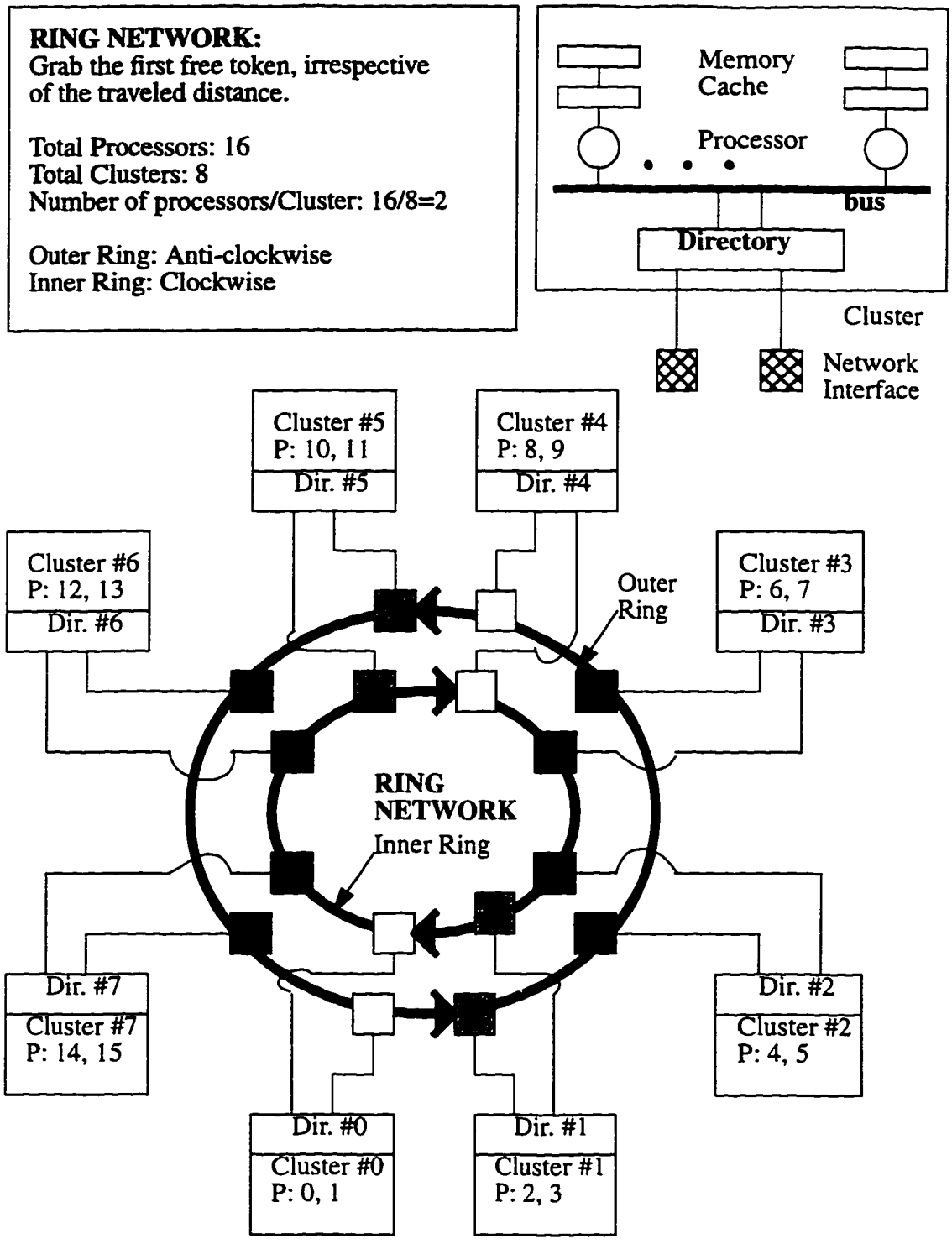


Figure 3.2b: Cluster-based multiprocessor (16 P, 8C) system using ring network

acknowledgment signal back to the source node, and cleans the link for future use. Delay due to each intermediate node is considered as 30 neno seconds.

3.2.2 Mesh Network Topology

Figures 3.3a and 3.3b show cluster-based multiprocessor systems using double connected mesh network for 4 clusters and 8 clusters, respectively. Each cluster is connected to other clusters with two communication channels, one for request and other one for acknowledgment. For simplicity, only one path is dedicated between any two clusters. Unfortunately it is not trouble free, there may be a common part of the channels being used by more than one pair of clusters. If every pair of clusters tries to communicate at the same time, one of them will be selected randomly and the other will wait in a queue. After first pair is done, second pair is selected randomly and the whole process repeats until every pair is done. X-Y routing strategy is followed for this network. Delay due to each intermediate node is considered as 30 neno seconds. This method decreases efficiency with increasing number of clusters, but for 4 to 16 clusters this effect is not tangible.

3.2.3 Hypercube Network Topology

Figure 3.4 shows cluster-based multiprocessor systems using double connected hypercube network for 8 clusters. A hypercube network for 4 clusters is the same as a mesh network for 4 clusters. Each cluster is connected to other clusters with two

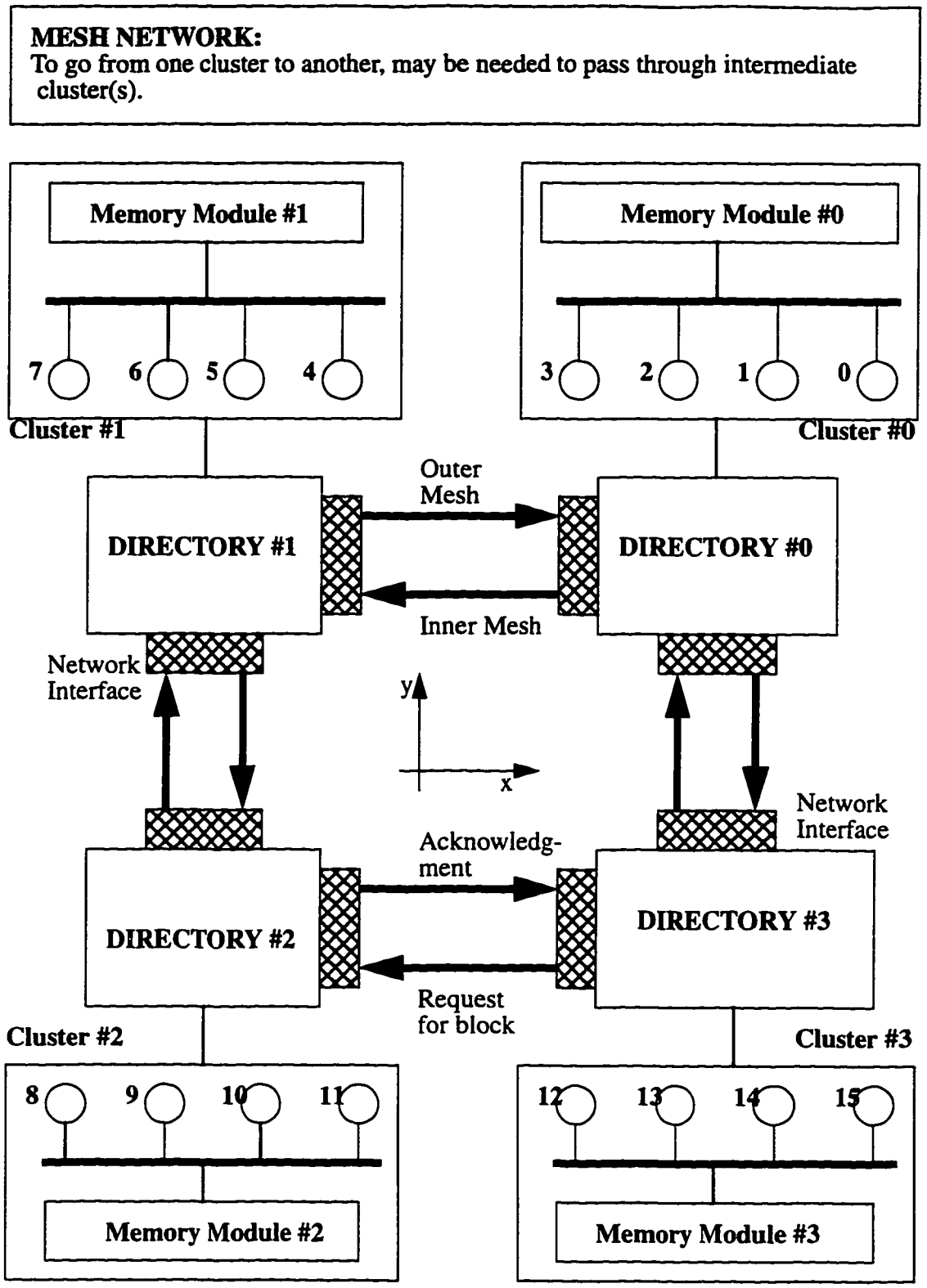


Figure 3.3a: Cluster-based multiprocessor (16 P, 4C) system using mesh network

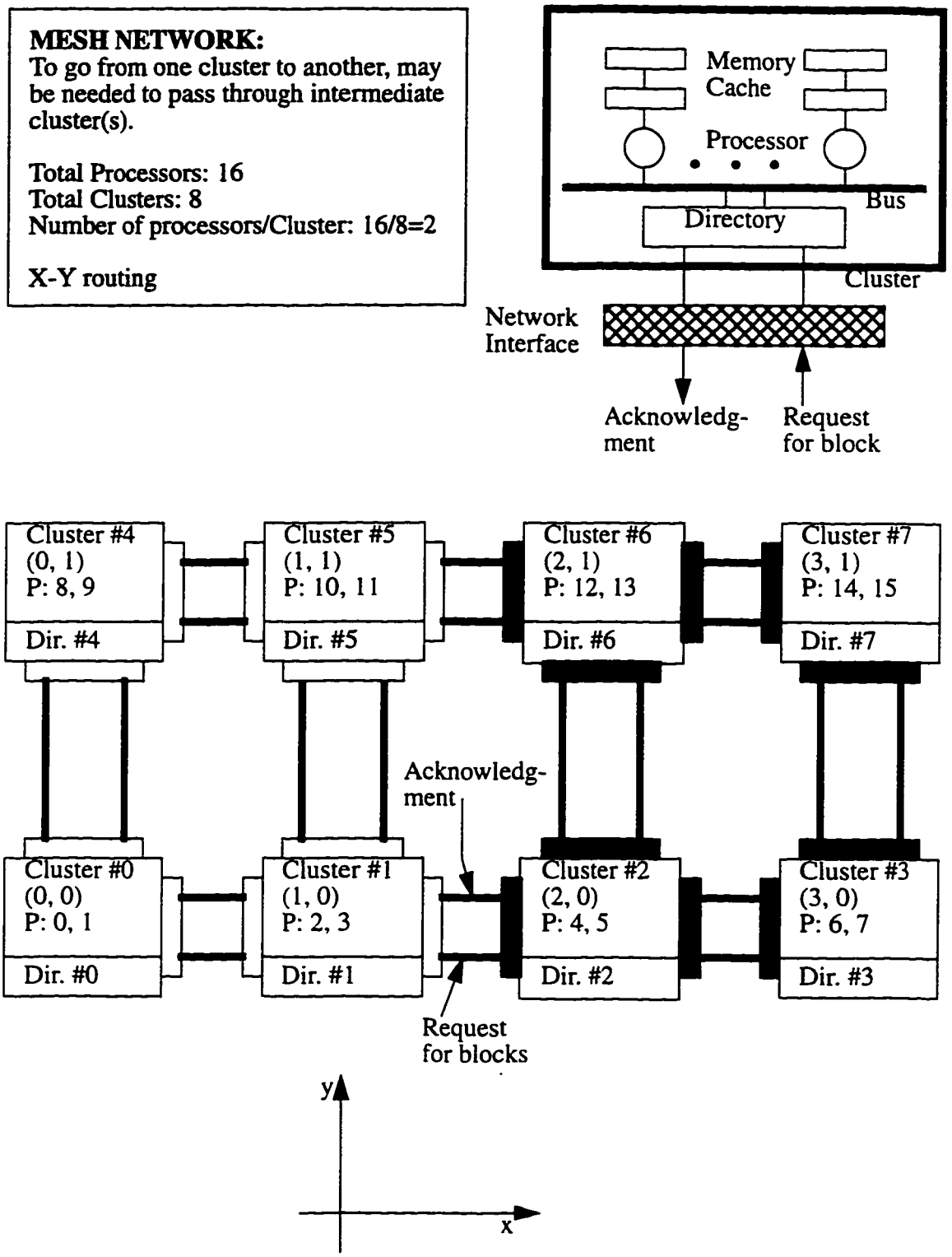


Figure 3.3b: Cluster-based multiprocessor (16 P, 8 C) system using mesh network

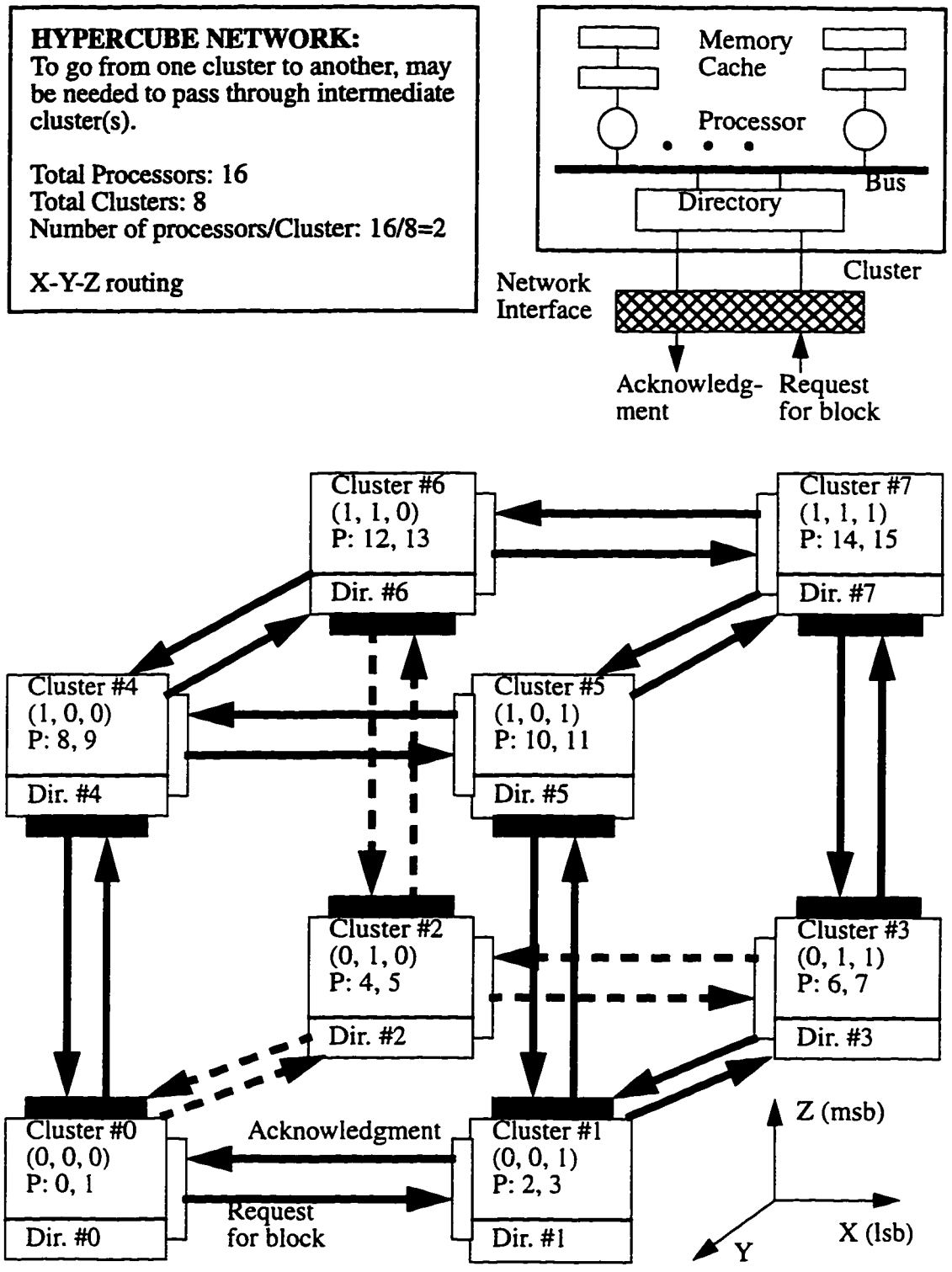


Figure 3.4: Simulation network with double connected hypercube network

communication channels, one for request and other one for acknowledgment. Here a 3-cube is considered and the clusters are arranged in a 3-dimensional (X-Y-Z) fashion. Each cluster in this system is connected to 3 other clusters. For simplicity, only one path is dedicated between any two clusters. This type of networks is not trouble free too, and there may be the same kind of link/channel sharing problem like mesh networks.

3.3 The cache-Coherence Protocol

Normal Read and Write operations are considered in this simulation work. MESI protocol is used to identify a cached copy and the states of directories. A memory location may be in one of these four states ---

a) Modified – the only valid copy in the whole system,

b) Exclusive – the only valid cached copy consistent with the memory copy,

c) Shared – the valid cached copies consistent with the memory copy, or

d) Invalid – the inconsistent (invalid) cached copy.

The memory system is broken into four stages of hierarchy, Two Levels of Processor Cache – Each processor has two caches, (i) first level cache and (ii) second level cache. Second level caches are larger than the first level caches. Processor caches are designed to match the processor speed and support snooping from/to the bus. (iii) Local or Home Cluster - This level includes some other processor's second level caches within the requesting processor's cluster and physical memory for some given memory addresses. The directory associated with a cluster is the owner directory for that cluster's memory addresses. Each directory has the information about all of its blocks if they are modified, exclusive, shared, or invalid. (iv) Remote Cluster Level - The fourth and final level for a memory block consists of the clusters marked by the owner directory as holding a copy of the block.

3.3.1 Memory Read Actions

When the requested block-address presents in the same processor's first level cache, the cache simply supplies the data and the read operation is done. If the read request is not satisfied from the first level cache, it is sent to the second level cache. No state change occurs at the directory level.

If the second level cache has a copy, then first level cache is filled from the second level, and the processor's request is satisfied. If the read request is not satisfied from the second level cache, it is sent to the local cluster level. No state change occurs at the directory level.

If local cluster has at least one cached copy of that memory block, the request is satisfied within the cluster and no state change is required at the directory level. If the request must be sent beyond the local cluster level, the owner directory for the memory location. If that block is un-cached, exclusive, or shared, main memory supplies that block. If that block is modified, the request is forwarded to the remote cluster indicated by the owner directory. Directory and maim memory states are updated accordingly.

3.3.2 Memory Write Actions

If the block is in the writing processor's cache and the state of the block is dirty, the write can be performed immediately. Otherwise, a Read-exclusive request is issued on the local cluster's bus to obtain exclusive ownership of the line and retrieve the remaining portion of the cache line.

If one of the caches within the cluster already owns the cache line, then the Read-exclusive request is serviced at the local level by a cache-to-cache transfer. Processors within a cluster are allowed to alternately modify the same memory block without any

inter-cluster interaction. If no local cache owns the block, then a Read-exclusive request is sent to the local cluster.

If the requested location is un-cached, exclusive, or shared the local cluster can immediately satisfy an ownership request. Also, if the requested location is in shared state, then all other cached copies must be invalidated. Invalidation requests are sent to those clusters that have a copy of that block, at the same time, the local cluster sends an exclusive data reply to the requesting cluster. If the directory indicates that the block is dirty, then the Read-exclusive request must be sent to dirty cluster.

If the required memory block is being shared, then the remote clusters receive an invalidation request to eliminate their shared copies; after receiving the invalidation, remote clusters send an acknowledgment to the requesting cluster. If the required block is dirty, the dirty cluster receives a Read-exclusive request. The remote cluster responds directly to the requesting cluster and sends a dirty-transfer message to the home indicating that the requesting cluster now holds the block exclusively.

When a writing cluster receives all the acknowledgments from the local or dirty cluster, it is obvious that all copies of the old data have been purged from the system. If the processor performs the write operation after receiving the acknowledgments, then the new value becomes available to all other processors at the same time.

3.4 Summary

A combination of snoopy and directory schemes is selected in this architecture to take advantages from both schemes. This architecture is a modification of DASH project to evaluate memory latency for different network topologies. Double ring, double

hypercube, and double mesh networks are used in this simulation program. Five SPLASH-2 applications, Water_sp, Ocean, FFT, and LU, are used and results are collected accordingly. Required data structures, described in next chapter, are maintained to store all necessary information during the simulation. To make comparison among interconnection networks, we have to follow some common assumptions, because each of these networks has its own characteristics. These assumptions, e.g., how this program works, how results are collected, and so on, are discussed in the following chapters.

Chapter 4

SIMULATION ANALYSIS

As already mentioned, the objective of this research work is to investigate overall memory latency of cluster-based cache-coherent multiprocessor system for different interconnection topologies namely double slotted ring, double connected hypercube, and double connected mesh network. Each network topology has its own characteristics, so in order to compare them it is obvious to follow some assumptions for every network. In Section 4.1 trace-driven simulation model is explained. Section 4.2 describes the SPLASH-2 applications used in this simulation. Section 4.3 shows different data structures required for this architecture. Section 4.4 is a collection of assumptions (MESI, cluster ownership, invalidation and update strategies, and so on) used. In Section 4.5, we explained how this simulation algorithm works. Simulation results are shown in Section 4.6. Finally, Section 4.7 is the summary of this chapter.

4.1 Trace-Driven Simulation Model

Trace-driven model, as shown in Figure 4.1a, has been used in this simulation. The simulator gets trace records from an application file. Each trace record has three fields: (i) requesting processor number, (ii) type of the operation (read or write), and (iii) memory address where read/write should be performed. Figure 4.1b is the flow-diagram of a cluster to show how do the processors work with trace file and network. Popular C programming language is chosen to write the simulator and it is executed on a SUN SPARC workstation.

4.2 Applications Used in this Simulation

Trace files used in this simulation are generated using MINT package designed by the Computer Science Department, University of Rochester, Rochester, NY[8]. They developed the techniques that improve locality of reference in parallel programs, so as to admit efficient execution on large-scale multiprocessors. Application trace files used in this simulation are Water-sp, Ocean, FFT, and LU [9].

Water_sp: Water_sp is molecular-dynamics code that computes the energy of a system of water molecules. This application computes the interactions between a set of water molecules over a series of time steps. For the problem size considered was 1728 molecules, each molecule interacts with all other molecules in the system. In this

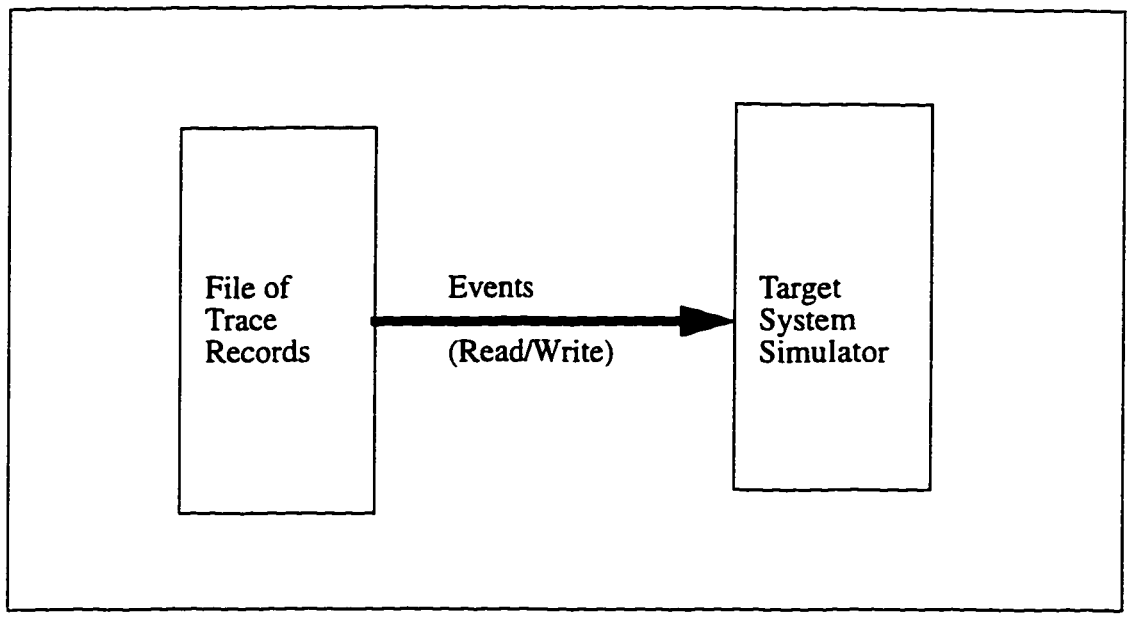


Figure 4.1a: Trace-Driven Simulation Model

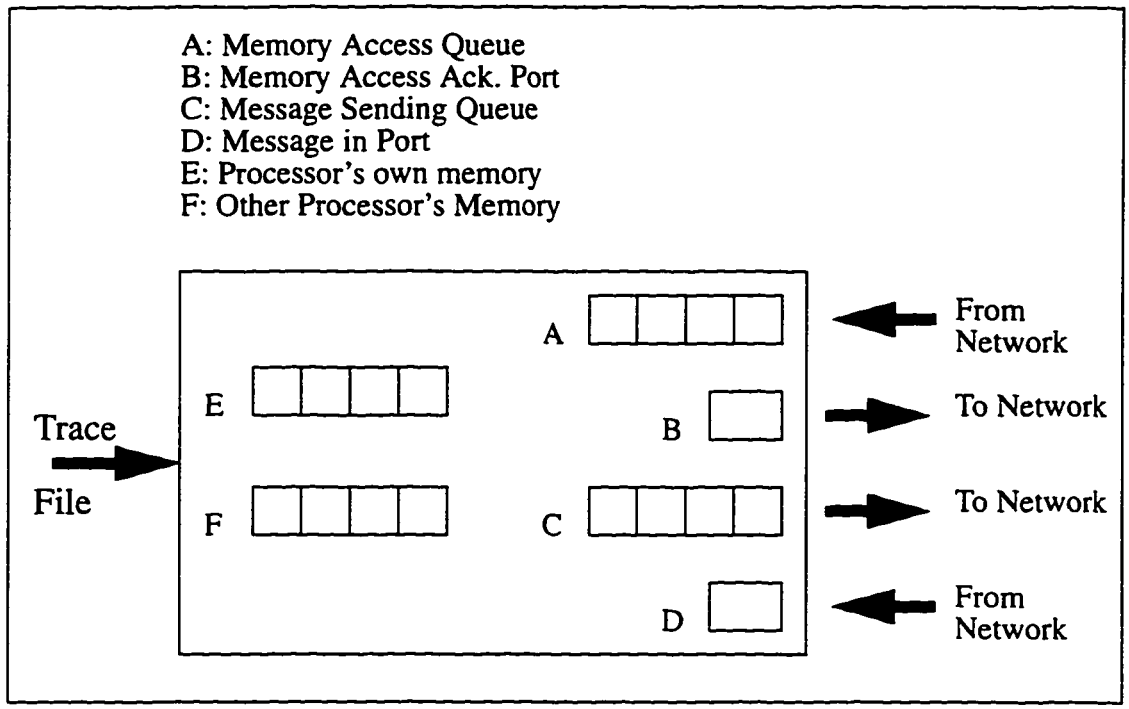


Figure 4.1b: Diagram of a Processor Node (Cluster)

simulation Water application achieves good speedup. The reason is that cache-locality is very high and the time between processor stalls indicates that Water is not highly sensitive to memory latency. The application computes the forces on molecules one at a time, and incurs misses only between force computations. The base problem size for an upto-64 processor machine is 512 molecules [13].

Ocean: The ocean application simulates large-scale ocean movements base on eddy and boundary currents. Contiguous partition allocation implements the grids to be operated on with three-dimensional arrays. The first dimension specifies the processor which owns the partition, and the second and third dimensions specify the x and y offset within a partition. The base problem size for an upto-64 processor machine is a 258x258 grid [13].

FFT: The First Fourier Transform (FFT) kernel implements a complex one-dimensional version of the six-step FFT algorithm described by Woo[13]. It is optimized to minimize inter-processor communication. The base problem size for an upto-64 processor machine is 65,536 complex data points (M=16) [13].

LU: The LU kernel factors a dense matrix into the product of a lower triangular and an upper triangular matrix. The factorization uses blocking to exploit temporal locality on individual sub-matrix elements. Non-contiguous block allocation prevents blocks from being allocated contiguously, but leads to a conceptually simple programming implementation. The base problem size for an upto-64 processor machine is a 512x512 matrix with a block size of B=16 [13].

4.3 Data Structures of this Architecture

The proposed architecture needs data structures for first level cache, second level cache, processor, processor-queue, memory, network-queue, and directory to perform the simulation.

For first level cache and second level cache, we need the information about memory block address and status.

4.3.1 Data Structure for Processor

Each processor is considered to have two levels of caches: cache level 1 (size may be 8 KB or 16 KB) and cache level 2 (size may be 256 KB or 2048 KB). Data structure for a processor is:

```
typedef struct proc Processor;
struct proc{
        cache1 cache_1[NO_BLK_CH1];
        cache2 cache_2[NO_BLK_CH2];
};

Processor processor[NO_PR];
```

Here, NO_BLK_CH1 and NO_BLK_CH2 are the total number of blocks in first and second level caches respectively, NO_PR is total number of processors, and cache1 and cache2 are two other structures with cache-block-address and cache-block-state.

After reading each trace from the trace file, we store it in the processor-queue under which it does belong. The structure of a processor queue is:

```
typedef struct proce_queue *Queue;
struct proce_queue{
        unsigned long addr;
        int     type_of_oper;
        Queue next_ptr;
};

Queue queueHeader[NO_PR];
```

Each processor has its own queue. After reading each trace from trace file the information is stored in the processor queue that is indicated by the processor number field of that trace record in order to maintain parallelism. When any of the queue is not empty, i.e., all processors have something to do, or any processor queue exceeds the maximum limit, the all the processors are allowed to perform their jobs (READ or WRITE operations).

4.3.2 Data Structure for Directory

Each owner directory keeps track of each memory block under its control. Checking the directory one can determine the state of a block, all sharing clusters if it is shared, the modified cluster if it is modified, and so on. The structure of a directory is:

```
tepedef struct dir Dir;
struct dir{
        struct block{
                unsigned long dir_blk_addr;
                int     dir_blk_state;
                int     shared_clus[NO_PR];
                int     mod_cluster;
        }blk[NO_MEM_BLK];
};

Dir directory[NO_CLUST];
```

Here, NO_MEM_BLK is the total number of memory blocks in one memory module and NO_CLUST is the total number of clusters in the system. For each cluster one directory is maintained and directory has the information of all blocks of that cluster. Before running the simulation program, the directory is initialized properly.

4.3.3 Data Structure of Network Queue

Operations that use the network to be performed are stored in a network queue to some time. The structure of the network queue is:

```
typedef struct net_queue *NetQue;
struct net_queue{
        short   loc_cluster;
        short   own_cluster;
        short   mod_cluster;
        NetQue          next_ptr;
};
```

Initially the network queue is empty. The operations, that need the network, are stored in the queue to satisfy the characteristic of multiprocessor system, These operations are performed simultaneously when the network queue is full to measure the network delay. Finally, the queue will be empty indicating that all of the operations that need remote resources are completed.

4.4 Assumptions

Assumptions are made to keep the simulation algorithm simple and specific. Sometimes there are more than one possible solutions for the same problem (e.g., when a

shared copy is written, the main memory copy may be updated or invalidated). For any of these cases if the same assumptions are not followed the ultimate result may vary and it may be very difficult to compare among different interconnection networks.

4.4.1 MESI: States of a Cached Copy

To identify any cached copy, the MESI cache coherence protocol is used. According to this protocol, any cached copy must be in one of the following states -

Modified: After more than one write on a cached block the main memory block is not updated, so main memory does not have a valid copy. The cached copy is the only valid copy of the block in the whole system. The block requires write back upon replacement. Any request for that block must be satisfied by the processor-cache that owns the modified block.

Exclusive: When a memory block is written for the first time, the main memory is also updated at the same time. So this cached copy is the only valid copy that exists along with the main memory copy. Any request for this block may be satisfied from the processor-cache that performed the write operation on that block or from main memory.

Shared: When a block is read from main memory, modified cluster, or from any other processor-cache who has a valid copy, the new state of that block is shared. So shared copy is a valid copy which exists with other copies including main memory copy. First time when a block is read is considered as a shared copy. The block does not require to be written back upon replacement.

Invalid: This block exists in a processor-cache but is not valid. When a write operation is performed on a block, that block becomes Exclusive or Modified, all other shared copies, if any, are invalidated to make sure that data from those blocks are not valid.

According to the assumptions, an exclusive copy is the only valid copy consistent with the main memory copy; also a shared copy may be the only valid copy with main memory copy. Read operation makes a block shared, but on the other hand write operation makes a block exclusive. Figure 4.2a is the state-transition graph for states (cached copies) of the adopted coherence protocol. P-Read and P-Write are the read and write operations, respectively, initiated by the same processor who has the block. Read-Blk and Write-Blk are Read and Write operations initiated by a processor who does not have a valid copy of the block. So after P-Read no state change occurs, after a Write-Blk the block must become invalid regardless of its original state, and so on.

Figure 4.2b shows the state-transition graph for states of transactions among the clusters. If the state of any block is changed by any operation, the owner directory must be updated. When a write operation is done into a modified copy within the same cluster, then no state change is needed. But after reading a modified block the directory must be updated. In this case main memory is also updated because the new state is shared. Also when a write operation is performed on a block owned by a remote cluster, necessary updates into caches, directories, and main memory are required. Initially all the blocks are considered to be un-cached.

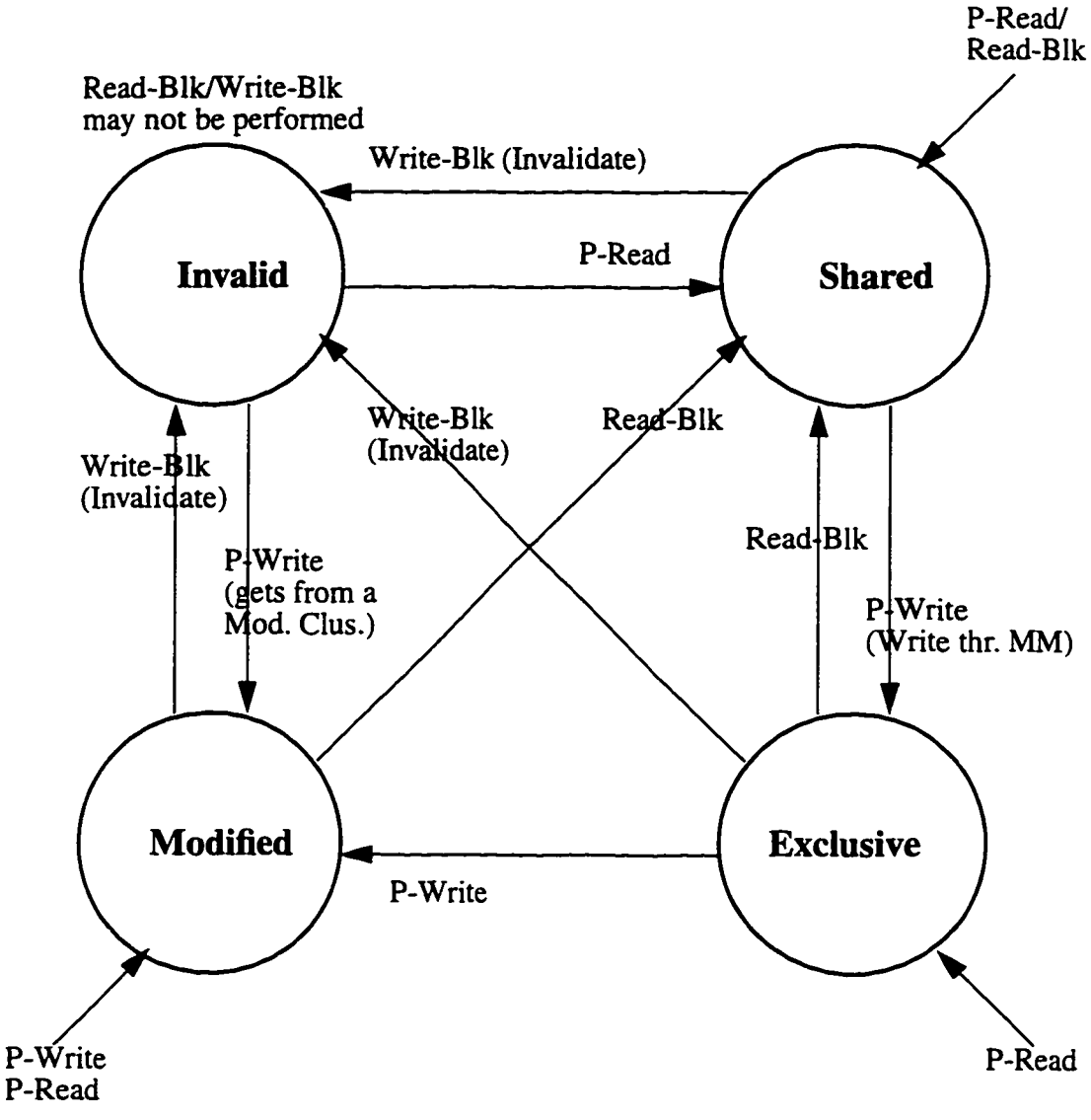


Figure 4.2a: State-transition graph for states of cached copies
P-Read and P-Write: Initiated by that processor which has the block
Read-Blk and Write-Blk: Initiated by other processor. .

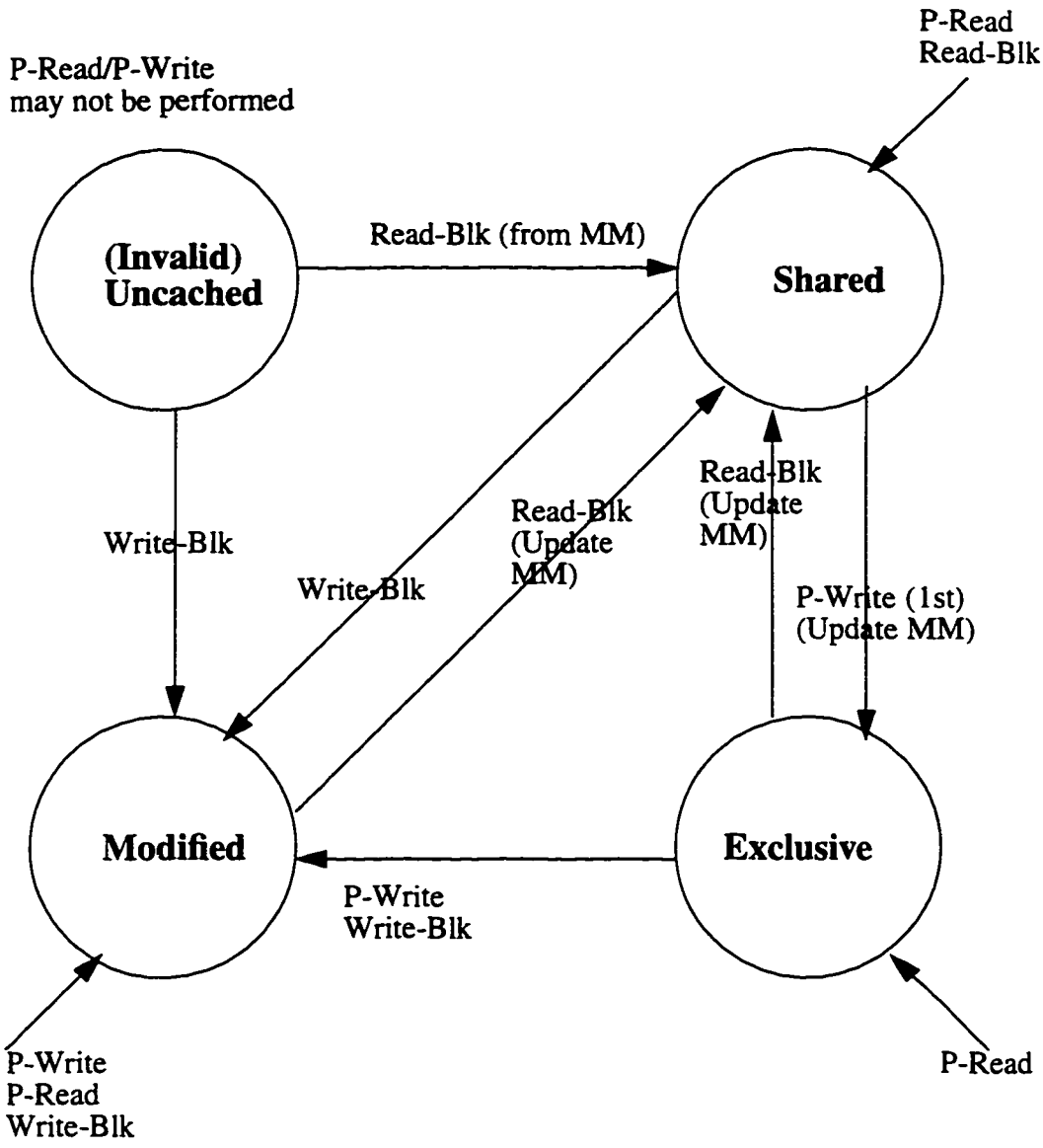


Figure 4.2b: State-transition graph for states of owner directory
Read-Blk and Write-Blk: Initiated by other processor.
Read-Blk and Write-Blk: Initiated by other processor.

4.4.2 Cluster Ownership of Memory Blocks

As shown in Fig. 3.1b for the architecture with 16 processors in 4 clusters, cluster 0 is the home cluster for processor 0, 1, 2, and 3. Directory 0 is the owner directory for cluster 0.

Table 4.1: Distribution of processors and memory among 4 clusters

Number of Processors	*Local Cluster*	*Owner Directory*	*Memory Module*
0, 1, 2, and 3	0	0	0
4, 5, 6, and 7	1	1	1
8, 9, 10, and 11	2	2	2
12, 13, 14, and 15	3	3	3

But for Fig. 3.1c, cluster 0 is the home cluster for only processor 0, and 1. Home cluster for processor 14, and 15 is cluster 7, and only directory 7 has the ownership for memory module 7, and so on.

Table 4.1 shows the distribution of the processors and memory modules among the clusters of a 4-cluster system. In this simulation, directory 0 contains information about memory blocks of memory module 0. Home cluster for processor 0, 1, 2, and 3 is cluster 0, and only directory 0 has the ownership for memory module 0, and home cluster for processor 12, 13, 14, and 15 is cluster 3, and only directory 3 has the ownership for memory module 3.

Table 4.2: Distribution of processors and memory among 8 clusters.

Number of Processors	*Local Cluster*	*Owner Directory*	*Memory Module*
0 and 1	0	0	0
2 and 3	1	1	1
4 and 5	2	2	2
6 and 7	3	3	3
8 and 9	4	4	4
10 and 11	5	5	5
12 and 13	6	6	6
14 and 15	7	7	7

Table 4.2 shows the distribution of the processors and memory modules among the clusters of a 8-cluster system. In this simulation local or home cluster for processor 14 and 15 is cluster 7, and only directory 7 has the ownership for memory module 7, and so on. If processor 1 wants to read a location owned by directory 3 and processor 12 has the modified copy of that block, then the interconnection network must be used to satisfy this request. When processor 1 fails to get the block from its local cluster, cluster 0 will communicated with cluster 3 to get the information about that block, and then with cluster 7 to access the data.

All clusters are equally spread and distance between any two consecutive clusters is 6 feet. Propagation delay is considered to be 1 nanosecond per foot. Transmission rate of each cluster is 1 GBps and processor speed is 166 MHz, i.e., 1 processor cycle takes 6 nanoseconds.

4.4.3 Cached Copies Invalidation and Main Memory Update Strategy

Two types of update policies already have been mentioned. To write in a shared memory block write-invalidate protocol is used. Before a write operation is performed on a shared block, an invalidate request is issued from the owner cluster to all sharing clusters, and all other copies are invalidated. Main memory copies are updated using write-update policy.

To insert a main memory block into first level or second level cache, a block is selected randomly if cache is already filled up, then the selected block is replaced by the new block. If the caches are not completely full, then just insert the new block to the first available free cache block.

4.4.4 Number of Processor Clocks Needed

Depending on from where the request is satisfied the number of processor clocks will be different. Table 4.3 shows different processor clocks required to find a block at different level (first level cache, second level cache, etc.). Whereas Table 4.4 shows different number of processor clocks needed to perform different events (snooping, broadcasting, etc.). Chart 1 shows Request Satisfied Vs the total number of processor cycles needed to satisfy the request. If the request is satisfied from the processor's first level cache, then the delay is considered to be 1 processor cycle, for second level cache it is 3 processor cycles, and so on.

Table 4.3a: Processor clocks required to find the block at different levels

No	*Memory Block is found*	*Number of Proc Clocks needed*
1	First Level Cache	1
2	Second Level Cache	3
3	Local Cluster	9
4	Local Memory	12
5	Remote Cluster	Vary

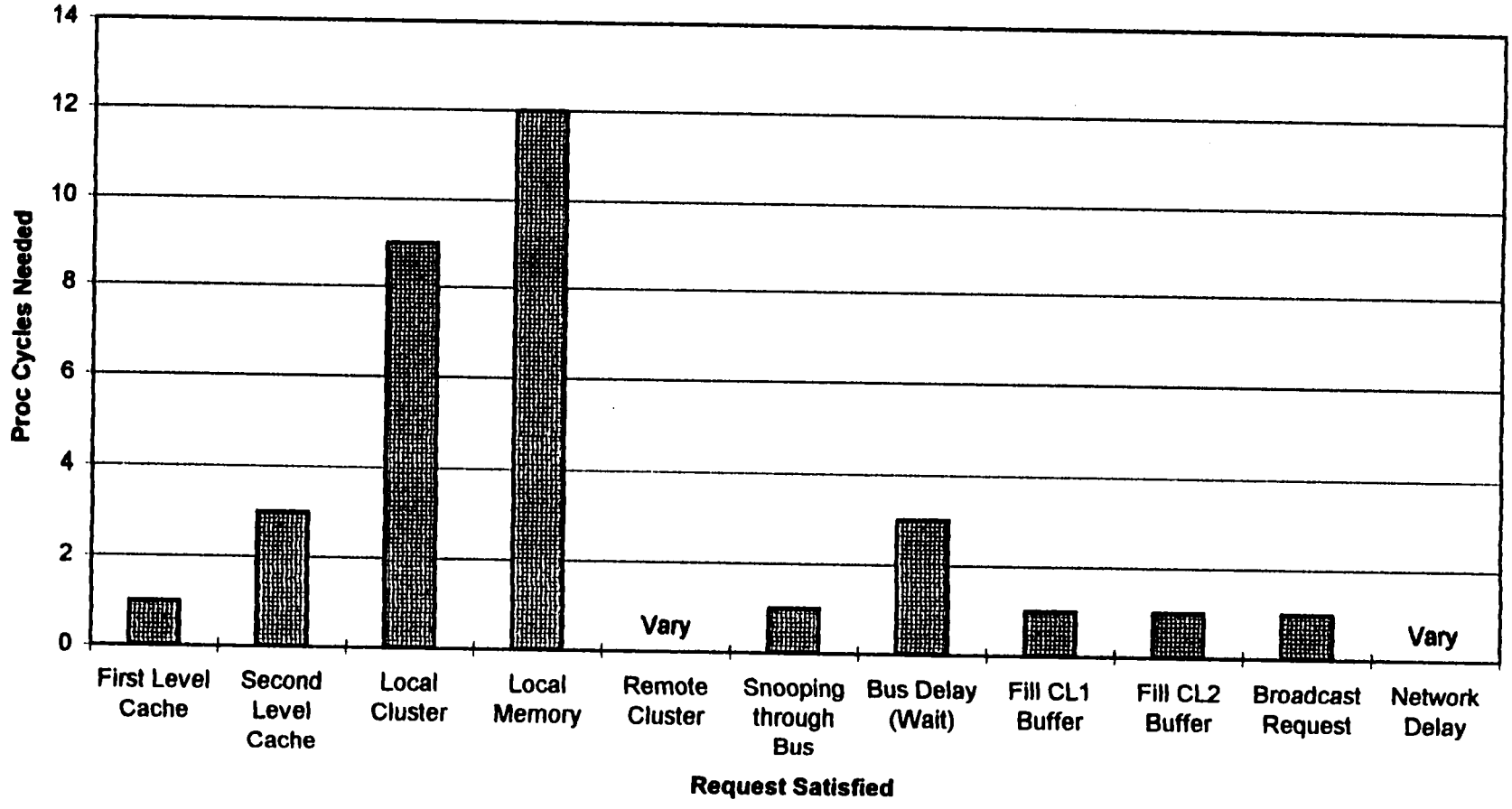
Processor speed is considered as 166 MHz, so time required for one processor clock is $1/166 \times 10^6$ = 6 nanoseconds.

Table 4.3b: Processor clocks required to perform different events

No	*To Perform The Event*	*Number of Proc Clocks needed*
1	Snooping through Bus	1
2	Bus Delay (Wait)	3
3	Fill cache Level 1 Buffer	1
4	Fill Cache Level 2 Buffer	1
5	Broadcast Request	1
4	Network Delay	Vary

All of the above assumptions are followed to simulate ring, mesh, and hypercube network. So the variation of overall memory latency will occur due to different characteristics of ring and mesh interconnection network.

Figure 4.3a: Requested resource Vs Processor cycles



4.5 Simulation Algorithm

The simulator is driven by suitable SPLASH-2 trace file. Each trace file has approximately 3.5 M of traces. Each trace record has three fields: requesting processor number, type of operation, and memory address. The requesting processor is interested to perform the operation on/from the memory location specified. Figure 4.3b is the flow-chart to explain the higher level algorithm:

START: (Higher-Level Algorithm)

1. Initialize the system (caches, memory modules, directories, etc.)

2. Read P traces from trace file (P = Number of processors)

    if Every processor has something to do, then

        a) Perform operations (Read or Write)

        b) Repeat from step 2;

    else if Any processor-queue exceeds the maximum allowed size, then

        a) Perform operations (Read or Write)

        b) Repeat from step 2;

    else (if end of trace file, then)

        Perform all remaining operations, if any;

3. Calculate overall memory latency;

END;

All the processors are allowed to perform their jobs at the same time. When all the operations are performed, the simulation program stops.

FLOW-CHART:
MEMORY LATENCY EVALUATION IN CLUSTER-BASED MULTIPROCESSOR SYSTEMS WITH DIFFERENT INTERCONNECTION TOPOLOGIES

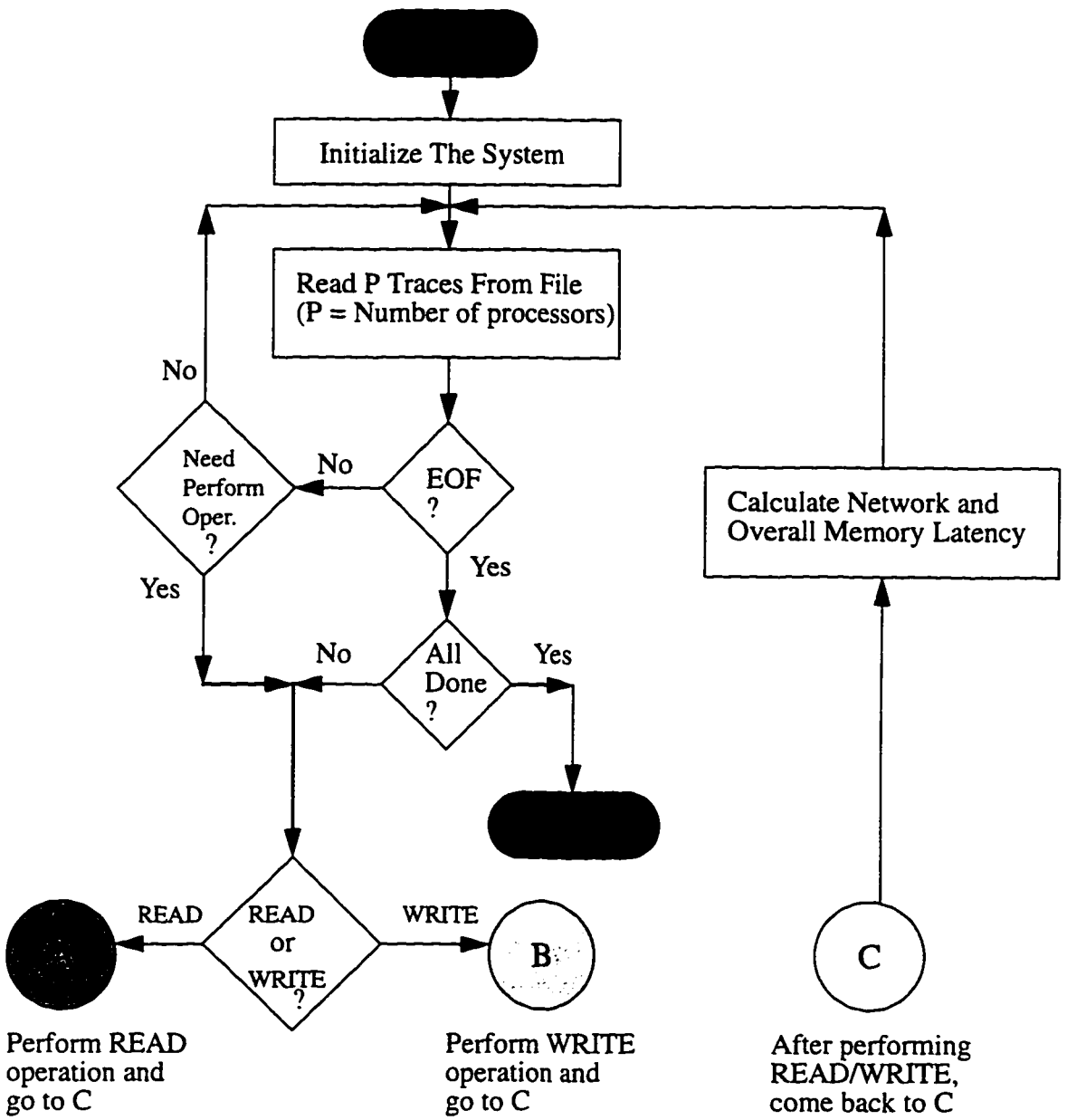


Figure 4.3b: Overall simulation flow-diagram

In the simulation, the main loop represents the parallelism. When every processor has something to do, they are allowed to perform their jobs simultaneously. Just to keep the processor queue size smaller than a certain number, processors are allowed to perform their jobs when one or more processor(s) exceeds that limit; at that situation it is also reasonable to assume that most of the processors have something to do. At this situation, one can expect almost accurate results.

Initialize The System: Before performing any trace operation, the whole system must be initialized properly. Initially no block is cached and main memory is the only source of data. Processor queue and network queue are empty. All caches, and directories must be initialized perfectly.

Read P Traces from trace file: If P is the number of processors in the system, then read P number of trace at a time. If the trace file is fully equally distributed, which means in any P number of consecutive traces no processor number is repeated, every processor will have something to do at this point. Otherwise, some processors will have nothing to do.

Need Perform Operation: If every processor has something to do, then let the processors perform their jobs simultaneously, otherwise read next P number of traces.

Check if All done: If all traces from trace file are performed and overall memory latency is calculated, then exit from this simulation. Otherwise perform operations.

Read or Write: Determine the type of the operation and perform the operation.

Calculate Network and Overall Delay: Must be some mechanism to calculate the network delay and the overall delay [15][16][17][18].

Figures 4.4a and 4.4b show how a read operation is performed. Only if the required block is not in First Level Cache (CL1), then Second Level Cache (CL2) is searched. If CL2 does not contain this specific block, then the requesting processor broadcasts a Read-Blk request within the cluster using the bus. All other processors (CL2 of each processor is connected to the bus directly as shown in Fig. 3.1a) of this cluster snoop on the bus. If any CL2 has this block will response and request will be satisfied locally. If more than one valid copy exists in this cluster, then request will be satisfied from the processor who will response first. If home cluster (or local cluster) fails to satisfy this request, then Read-Blk request is forwarded to the owner directory of required block as shown in Figure 4.3b. Directory state of this block may be one of the following,

1. Modified,

2. Exclusive or Shared,

3. Invalid (or Un-cached)

Modified Block: This is the only valid copy in the entire system, main memory copy is already invalidated. So, any request for this block must be satisfied from the modified cluster or processor.

Exclusive or Shared Block: Exclusive copy (the only valid copy) and shared copies are consistent with the main memory copy. In this case request is satisfied from the main memory with the permission of the owner directory.

Invalid or Un-cached Block: Block that never has been cached or has been invalidated by any processor. This block can not be accessed.

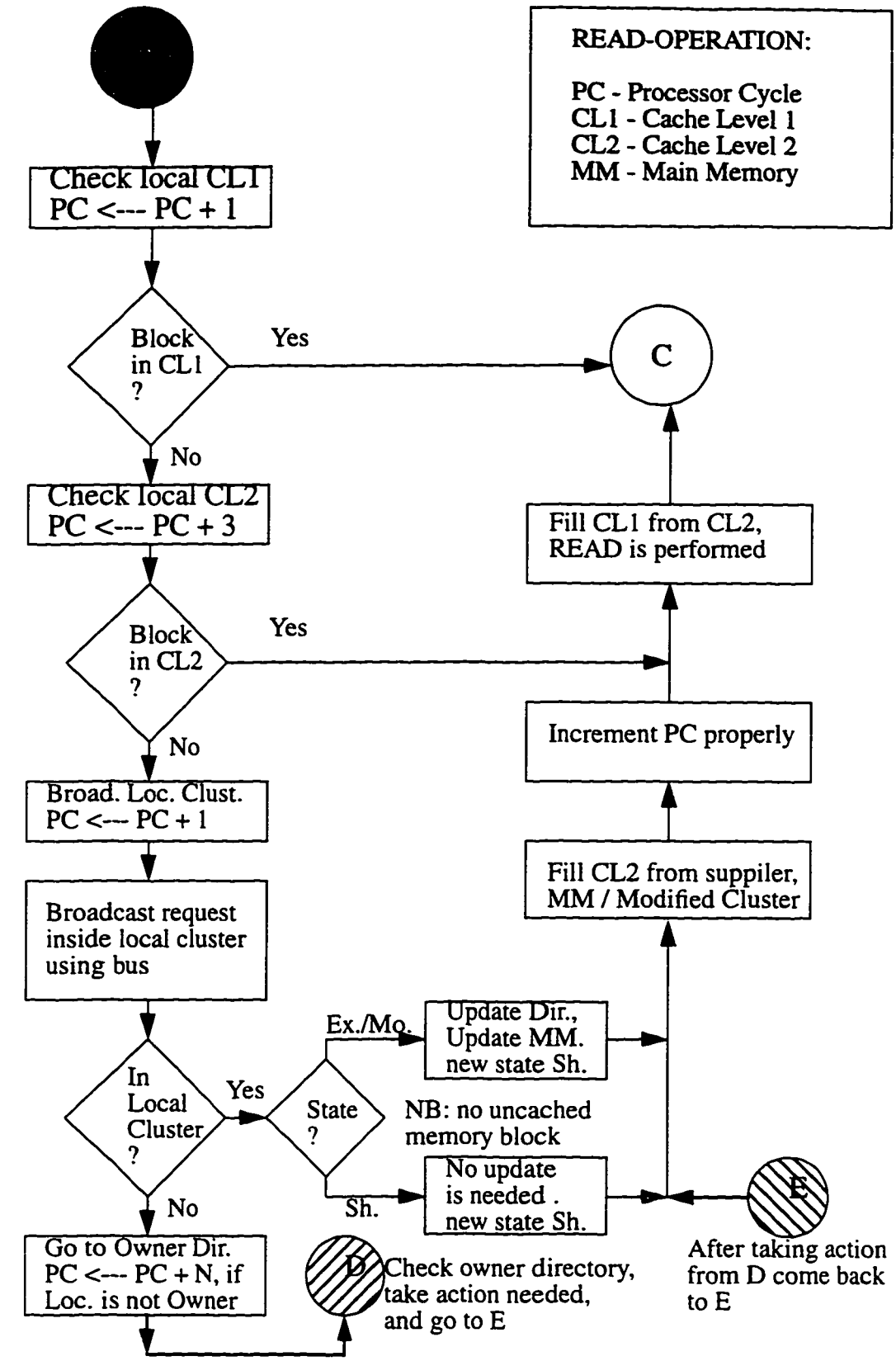


Figure 4.4a: Flow-chart for Read operation (part I)

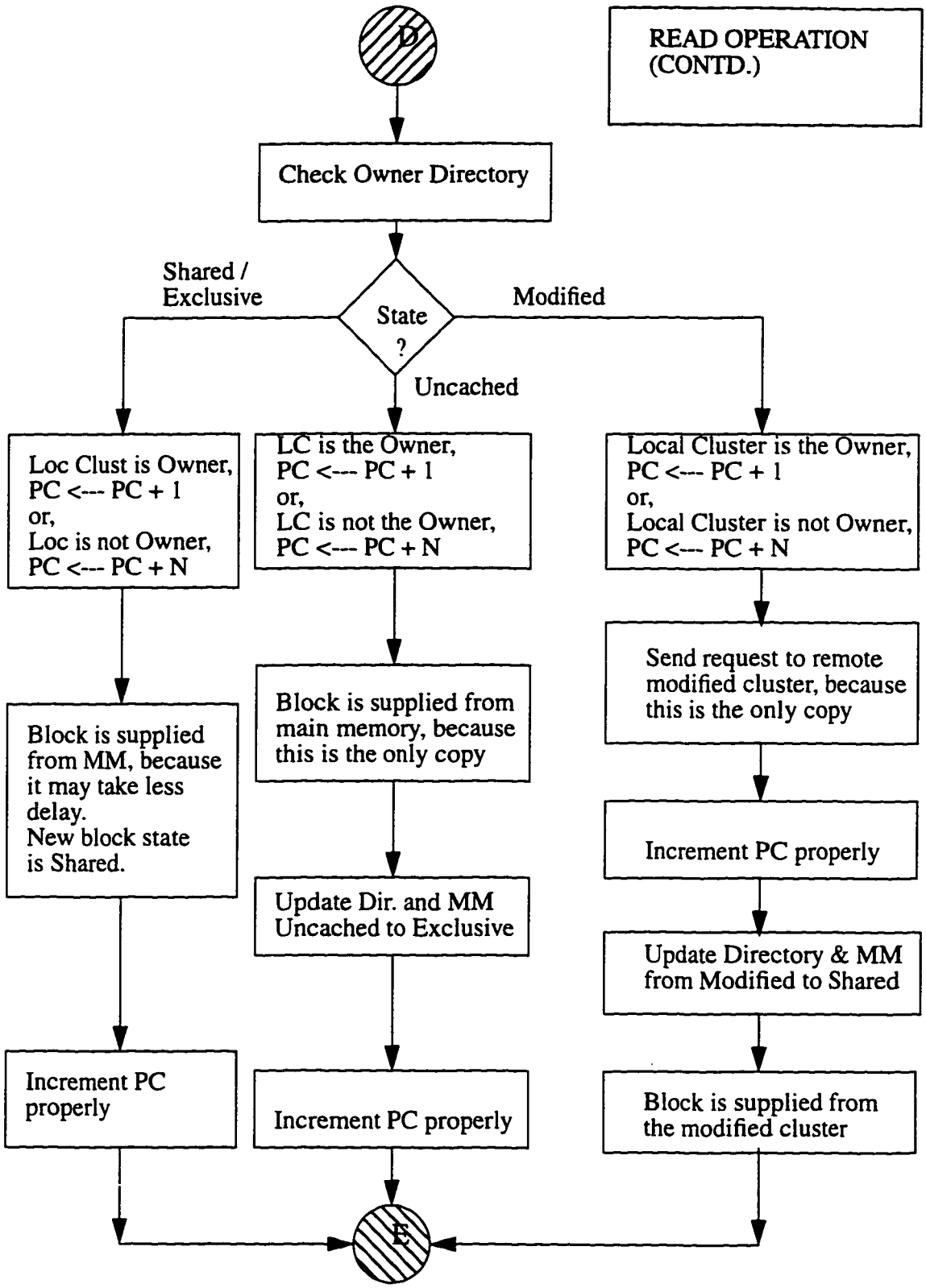


Figure 4.4b: Flow-chart for Read operation (part II)

If block is modified, then modified cluster will satisfy the request simply because, this is the only valid copy in the whole system. Main memory needs to be updated as new state of this block is shared. If the block is exclusive, shared, invalid, or un-cached, then request will be satisfied form main memory to make the algorithm easy to implement, moreover this may reduce overall communication delay. Directory is updated accordingly, so owner knows most recent information about each block under its control.

Figures 4.5a, 4.5b, and 4.5c are the flow diagrams to explain how a write operation is performed. Basic idea is similar as read operation, but write operation is more complex when compared with read operation. Main two reasons behind this phenomena are:

1. When a block is found in CL1 and/or CL2, immediately read operations can be performed. For write operations situation is different, simply because the state of that block changes. If that block is in shared state, then situation is more complex. All other copies must have to make invalid before writing, and to invalidate a copy traveling through the whole network may be needed. Also main memory and directory may have to update.

2. When a shared block is read, the new state is also shared and no invalidation is required. But in case of write on a shared copy, new state is exclusive and all other shared copies must be invalidate before writing using write-invalidate consistency command. Moreover main memory is updated using write-update policy.

It is easy to realize that, write operations need more work, and it is obvious that delay for a write operation is more than a read operation even though the block is found in same distance.

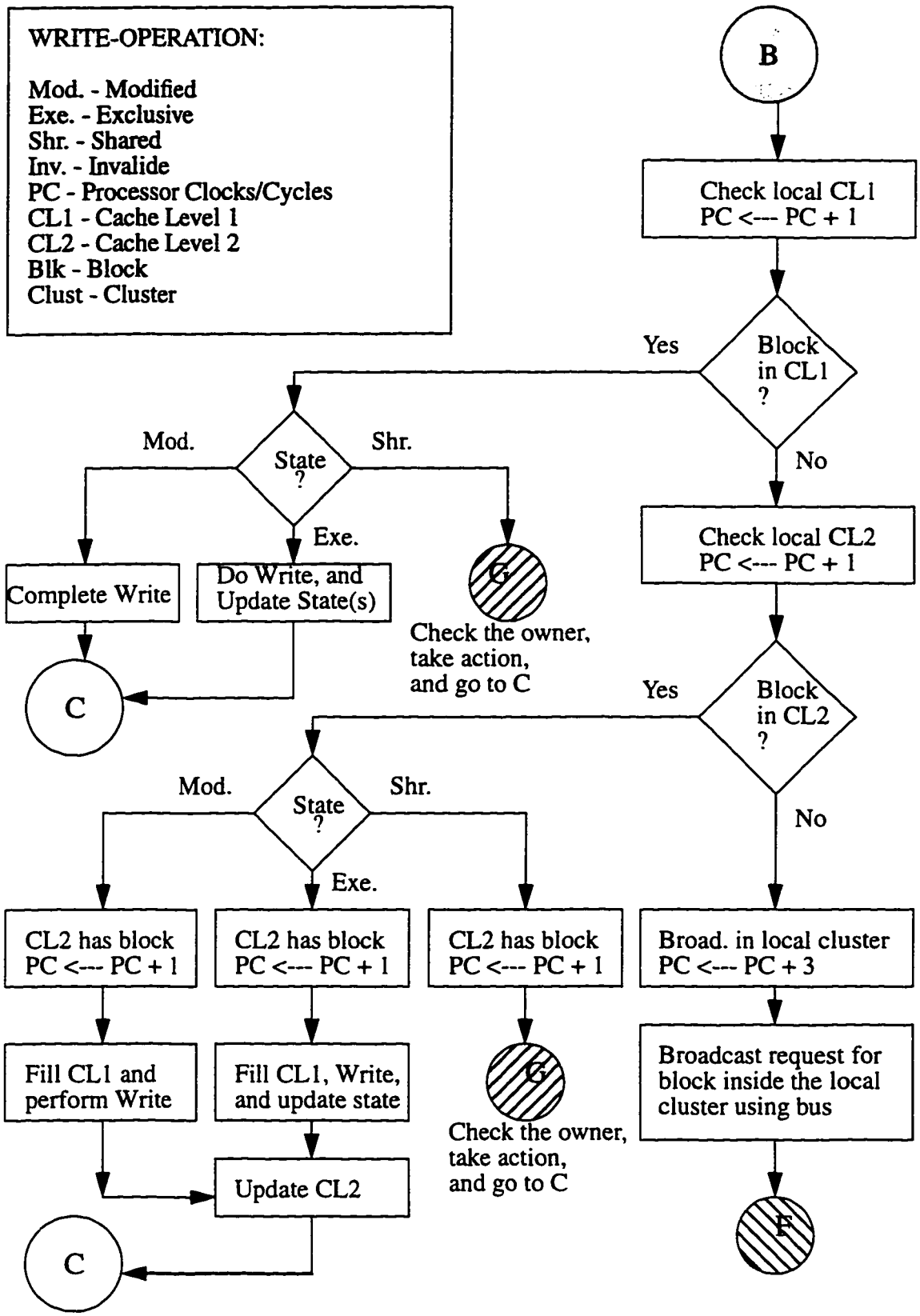


Figure 4.5a: Flow-diagram for Write operation (part I)

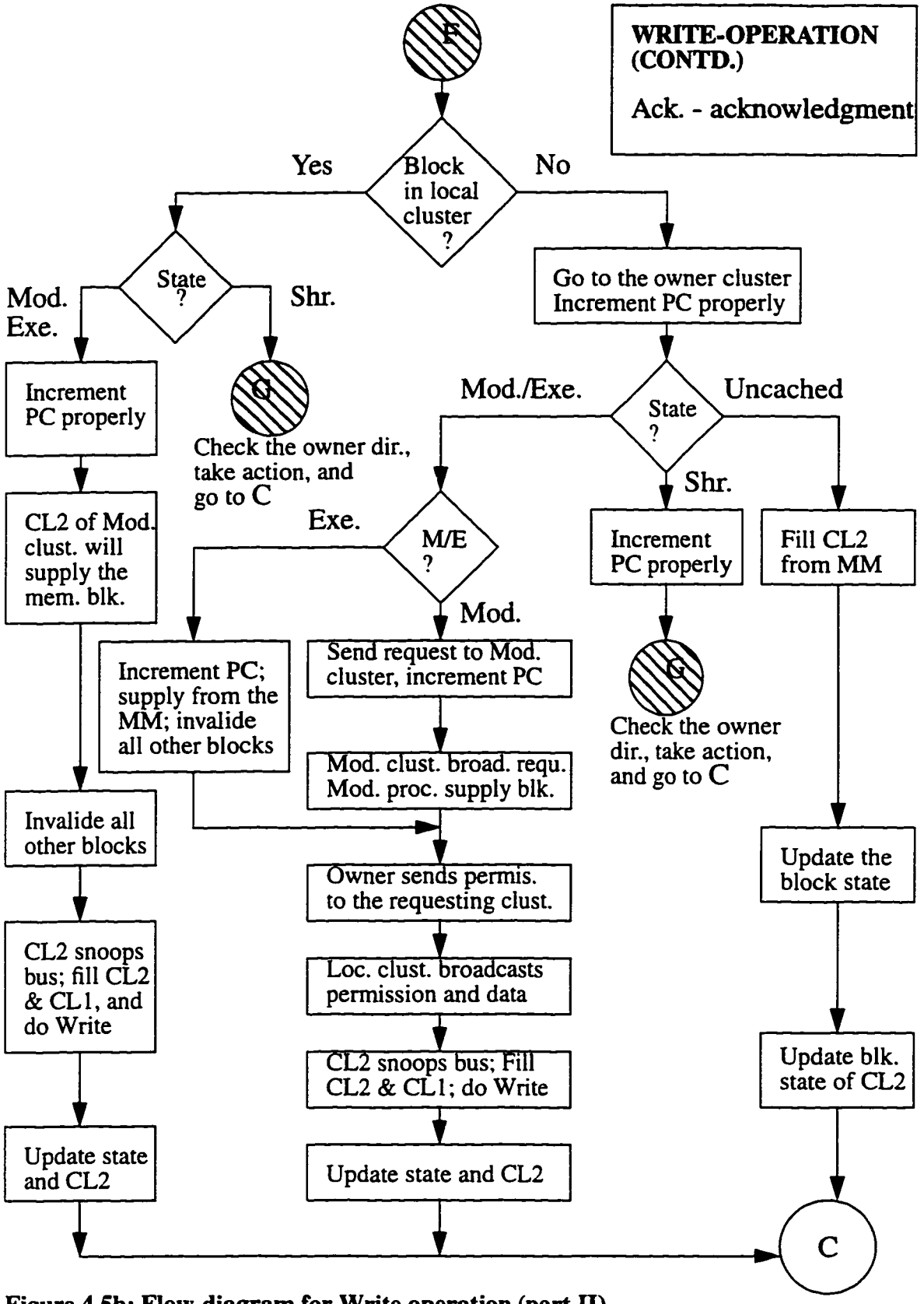


Figure 4.5b: Flow-diagram for Write operation (part II)

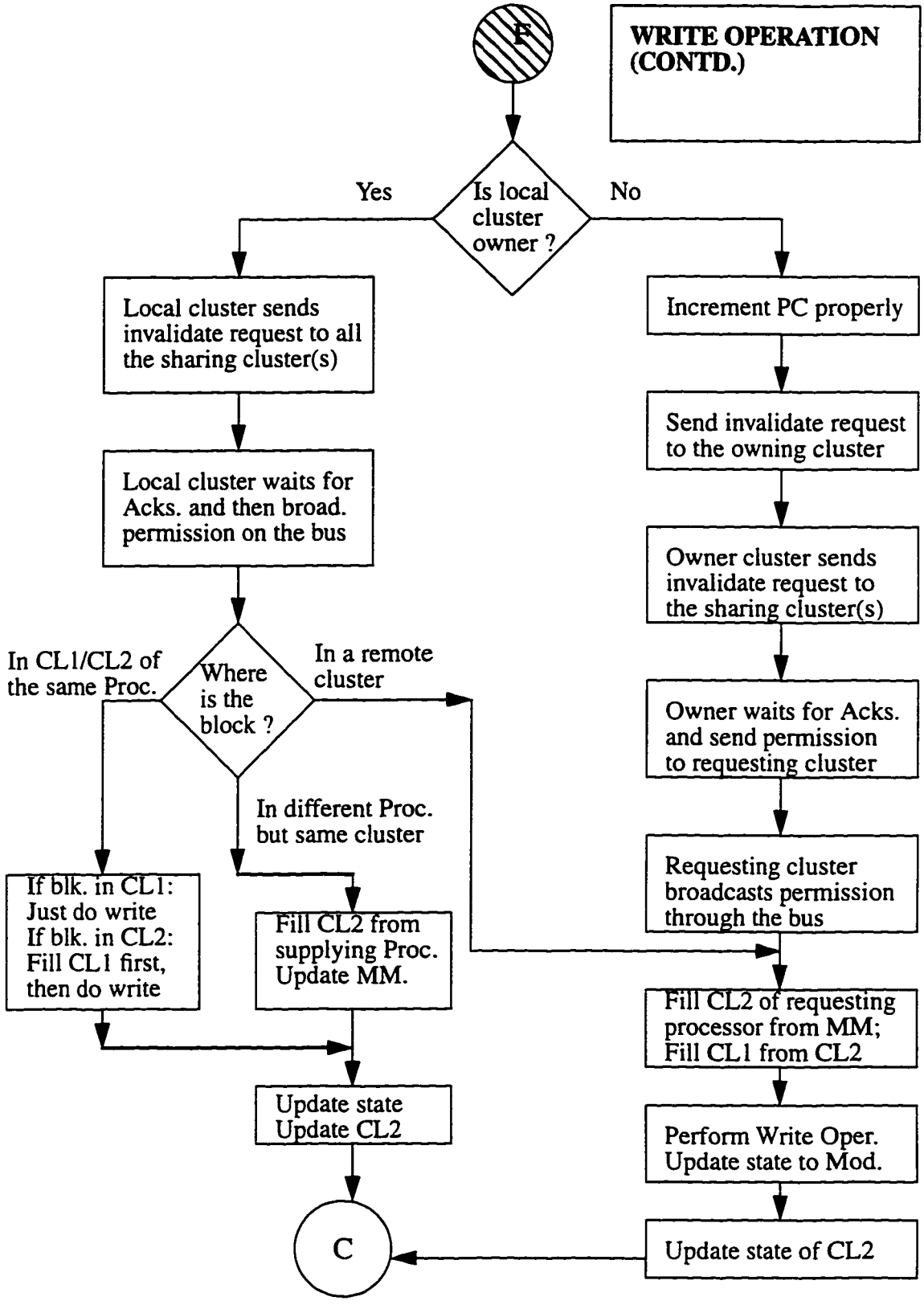


Figure 4.5c: Flow-diagram for Write operation (part III)

Figure 4.6 shows the execution procedure in proposed (double) ring network. One ring, say the inner ring (RING1), rotates in clockwise; the other ring, the outer ring (RING0), rotates in counter-clockwise. The shortest-path strategy (RING0 or RING1) is used by the cluster to transfer information or acknowledgment to other clusters. Each cluster is connected to both. 1 processor clock = 6 neno seconds = 6/1,000,000,000 Sec Transmission rate = 1 GBps = 8,000,000,000 bps. Propagation delay between two consecutive clusters is = 6 feet * 1 neno second = 1 processor clock. Delay due to READ block transfer = 32*8bits/1GBps = 5 processor clocks. Delay due to WRITE block transfer = (32+4)*8bits/1GBps = 6 processor clocks. Delay due to each intermediate node = 30 neno seconds = 5 processor clocks. Network delay = prop delay + message delay + intermediate node delay

Figure 4.7 explains how the proposed mesh network works. Each cluster is connected to other clusters by two communication channels, one for request and other for acknowledgment. To find out the destination message first travels in x-direction then y-direction. To reach at the destination from the source, message may pass through intermediate cluster(s). More than one pair of clusters may require the same portion of communication channel. In this case FCFS policy is used. If more than pair needs the same path at the same time, then randomly one pair will be selected to access the path, and other pairs will wait in a queue for next time. The delay from one cluster to the next (in any direction) is 10 micro seconds, i.e., 1667 processor cycles.

Here overhead delay is considered and the delay for actual memory block and/or acknowledgment is not measured, because they are almost the same for each network.

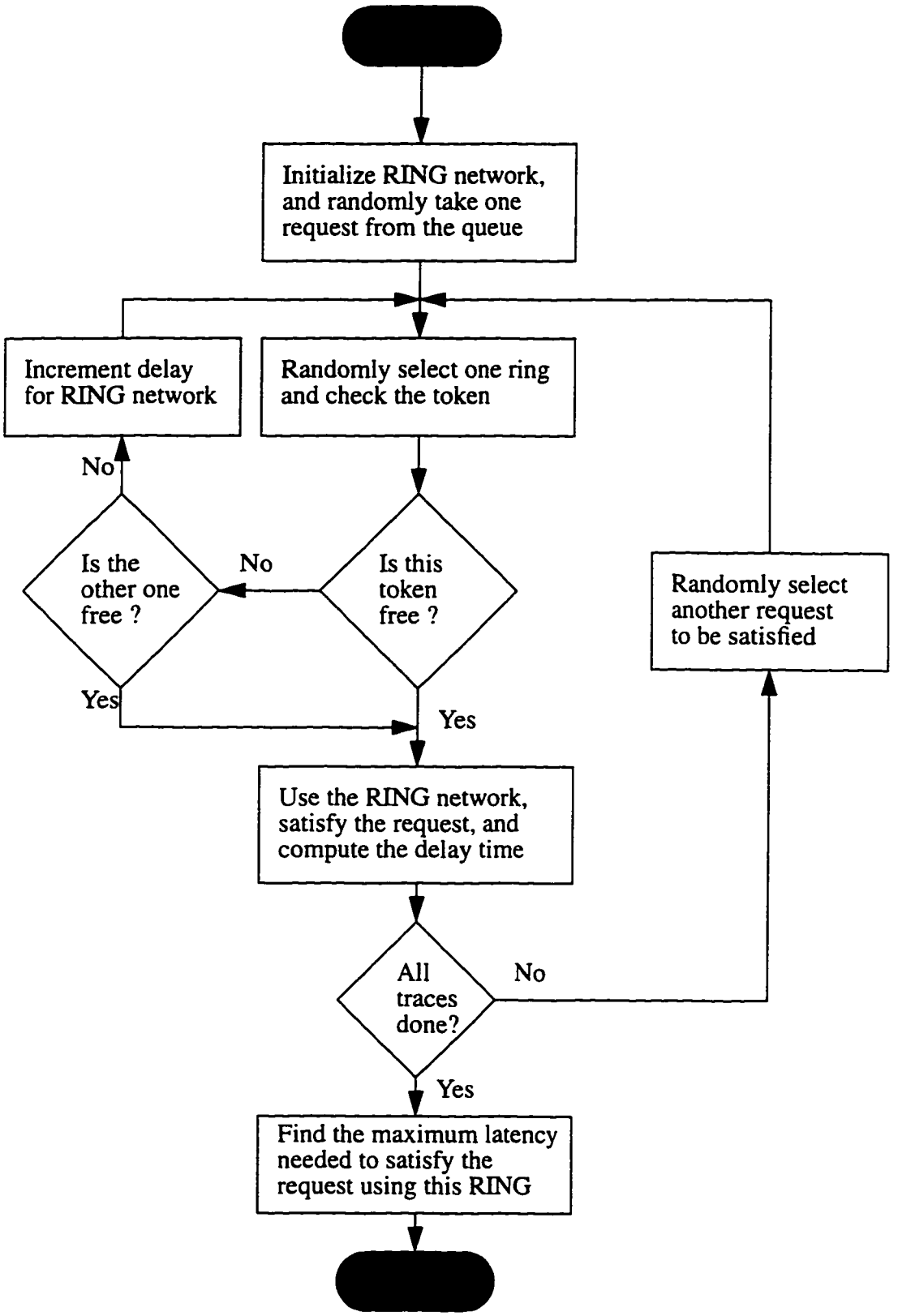


Figure 4.6: Flow-chart for ring network

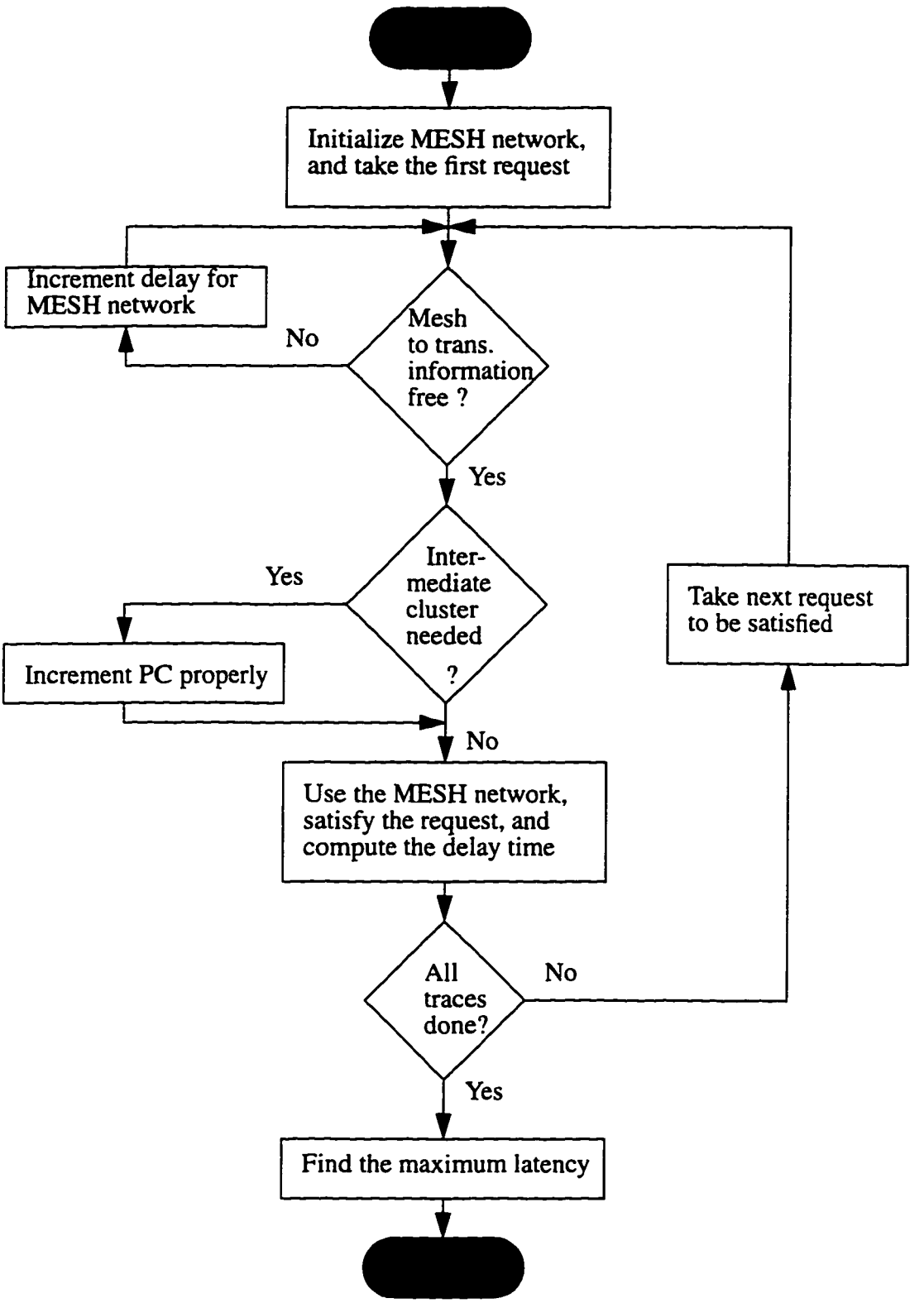


Figure 4.7: Flow-chart for mesh network

Figure 4.8 is the flow diagram to show how a hypercube network works. To find out the destination message first travels in x-direction, then in y-direction, and finally in z-direction. To reach at the destination from the source, message may pass through intermediate cluster(s). The delay from one cluster to the next (in any direction) is 10 micro seconds, i.e., 1667 processor cycles.

This simulation is concerned about overall memory latency. For a shared memory multiprocessor system the interconnection network has a great effect on the memory latency [19]. This algorithm helps to compare total delay time needed for a particular application but different interconnection network.

4.6 Simulation Results

In any multiprocessor system, the overall latency is dependent on the interconnection network. Here double connected ring, double connected mesh, and double connected hypercube networks have been considered.

To collect the results, the following procedure is used,

The executable file (say, simu) is ready to be used to collect results

1. Select one trace file (the application to be tested, may be water_sp)
2. Run the executable file using the selected trace file. e.g.,

    [sunrise]simu water <enter>

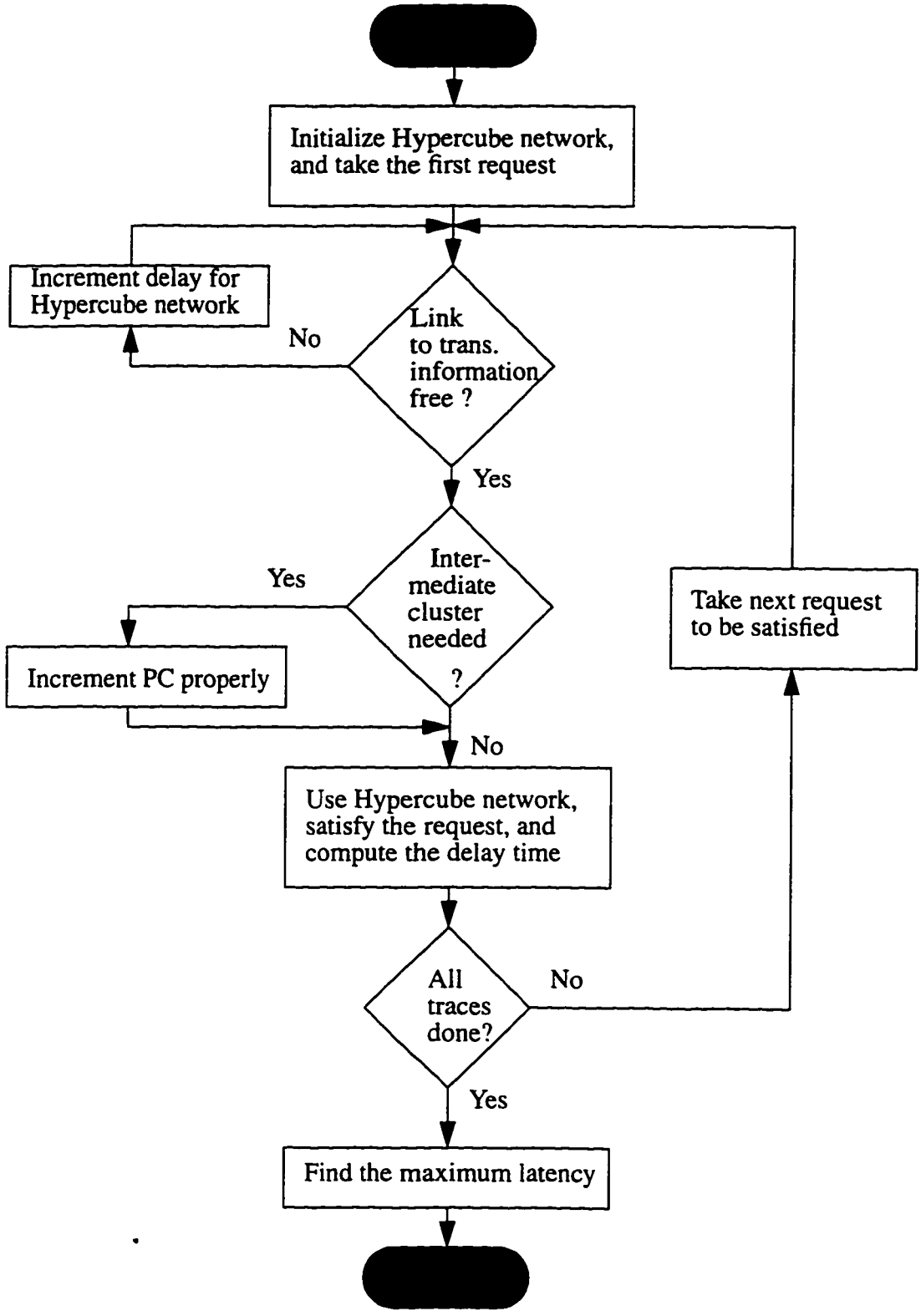


Figure 4.8: Flow-chart for hypercube network

3. During the runtime the program will ask for some information such as name of the network to be used, name of the machine being used, today's date, and so on. e.g.,

[sunrise]Enter network name to be used (ring/mesh/hypercube) >

To select ring network type: ring <enter>

4. Program will collect all necessary information, calculate the results, save the result in a file (say, simu.out), and display a message for the user. e.g.,

•

[sunrise]Enter network name to be used (ring/mesh) >ring

MESAGE:
Simulation is done successfully. Results are in simu.out file.

5. Open simu.out to see the results. A sample result file:

*** SIMULATION IS DONE SUCCESSFULLY ***

Network used: ring SunStation: sunrise
Today's date: 10/17/96 Trace file: radix

Total number of proc. clocks are: 8968 K
Network delay (proc. clocks) are: 242 K

Total Processors: 16 Total Clusters: 8
CL 1 Size: 8 Kbytes CL 2 Size: 256 KB

Read + Write = 2216296 + 1283579 = 3499875
Total number of CL1 read hit is.....= 2216083
Total number of CL1 write hit is.....= 1278498

Calculation:

Percentage of Read-Hit: 99.99
Percentage of Write-Hit: 99.60

.o0OO0o..o0OO0o..o0OO0o..o0OO0o..o0OO0o.

Here, CL1 _hit means the request (read or write) is satisfied from the first level cache of the requesting processor. So, CL1 _miss means the request is not satisfied from the first level cache of the requesting processor. Of course, CL1 _hit operation takes the east amount of time to be satisfied.

The term Total number of processor clocks (or, total delay) indicates the total amount of time required to execute the whole trace file, that simply means,

Total delay = Cluster delay + Network delay, where cluster delay is the time required to check CL1, CL2 (if needed), local cluster (if needed broadcast request), owner cluster (if needed where local cluster is the owner) without using network, and Network Delay is the is the delay time required to reach owner cluster(if needed when local cluster is not the owner) using network, modified cluster (if needed) using network, and invalidate shared copy (if needed) using network, update main memory copy (if needed) using network, and so on.

Also it is clear that,

Cluster delay = f(trace file), and

Network delay = f(trace file, network)

If we keep the trace file fixed and change the interconnection network, then cluster delay is a constant. Now if this program is run for two different networks (ring and mesh) using the same trace file, then the difference between two results (Total-delay for ring ~ Total-delay for mesh) is the measure of this two networks.

4.7 Summary

In some situations, for the interconnection networks, there may be more than one solution. For every case if the same assumptions are not followed the ultimate result may vary and it may be very difficult and not reasonable to compare among different networks. MESI (Modified/Exclusive/Shared/Invalid) protocol is used to identify any cached copy. To write in a shared memory block write-invalidate policy is used. Only main memory is updated using write-update policy. The simulator is driven by a trace file. A higher level algorithm has been developed where the main loop represents the parallelism. Simulation results are collected to compare among slotted double ring network, double connected mesh network, and double connected hypercube network. The simulator gives the network delay and the overall delay for each network. Next chapter discusses the simulation results.

Chapter 5

SIMULATION RESULTS AND DISCUSSIONS

In this research work, overall memory latency for a multiprocessor system with 16 processors has been measured for three different interconnection networks: ring, mesh, and hypercube. Two types of system configurations are implemented: 4-cluster (4 processors in a cluster) and 8-cluster (2 processors in a cluster) multiprocessor. Any two consecutive clusters are considered to be 6 feet apart and propagation delay is 1 nanosecond per foot. The processor speed is 166 MHz, the transmission rate of a cluster is 1 GBps. Each processor has two levels of caches: cache level 1 with a size of 8 KB (or 16 KB) and cache level 2 with a size of 256 KB (or 2048 KB). Five different applications are used in this simulation. Section 5.1 contains the information about trace files. Results for 4 cluster systems and 8 cluster systems are represented in Section 5.2 and 5.3, respectively. Section 5.4 is the performance analysis of ring, mesh, and hypercube network topologies with variable cache sizes. All three networks has been considered with variable cache sizes, because ring gave better performance for all applications. Finally, Section 5.5 is the summary of this chapter.

5.1 Applications: Read and Write Operations

Five different applications are used: Radix, Water_sp, Ocean, FFT, and LU. Total number of operations of LU application is 3499999, whereas each of the other applications has 3499875 operations, as shown in Table 5.1a.

Table 5.1a: Information about different applications: Total operations

DIFFERENT APPLICATIONS	*TOTAL OPERATIONS*	*TOTAL READ OPERATIONS*	*TOTAL WRITE OPERATIONS*
Radix	3499875	2216296	1283579
Water_sp	3499875	2449022	1050853
Ocean	3499875	2305809	1194066
FFT	3499875	2369799	1130076
LU	3499999	2360376	1139623

The total number of traces for any application is almost the same, but Read-hit and/or Write-hit are different. According to Table 5.1b, Radix application gives more cache level 1 Read-hit (99.99%) and Write-hit (99.60%), which means more processors request for a block that is in its own cache level 1. On the other hand, LU application gives the lest cache level 1 Read-hit (81.30%), and FFT gives the lest cache level 1 Write-hit (33.34%).

Table 5.1b: Information about different applications: Read-hit/Write-hit

Different Applications	*% CL1 Read-hit (8 clusters)*	*% CL1 Write-hit (8 clusters)*
Radix	99.99	99.60
Water_sp	99.19	98.81
Ocean	89.25	89.26
FFT	81.98	33.34
LU	81.30	34.78

5.2 System with 4 Clusters

In a 16-processor and 4-cluster multiprocessor system, (4 processors in a cluster), Radix application needs the smallest overall and network latency, and LU needs the largest overall and network latency as shown in Table 5.2 when clusters are connected through a ring network. These results are collected for a system with cache level 1 8 Kbytes and cache level2 256 Kbytes.

Table 5.2: Overall delay/Network delay - Ring Network (4 clusters)

No	*Applications (trace files)*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Radix	8849	86
2	Water_sp	8962	95
3	Ocean	9934	106
4	FFT	11392	122
5	LU	11673	134

If ring is replaced by a mesh network (or a hypercube network), the multiprocessor system performance remains the same as shown in Table 5.3. This is obvious because of the architecture of 4-cluster system.

Table 5.3: Overall delay/Network delay - Mesh Network (4 clusters)

No	*Applications (trace files)*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Radix	8849	86
2	Water_sp	8962	95
3	Ocean	9934	106
4	FFT	11392	122
5	LU	11673	134

Ring, Mesh, and hypercube network give the same results when 4 clusters are used to design a multiprocessor system.

Overall delay and Network delay for Radix application are shown in Table 5.4.

Table 5.4: Overall delay/Network delay - Radix (4 clusters)

No	*Network used*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	8849	86
2	Mesh	8849	86
3	Hypercube	8849	86

Overall delay and Network delay for Water_sp application are shown in Table 5.5.

Table 5.5: Overall delay/Network delay – Water_sp (4 clusters)

No	*Network used*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	8962	95
2	Mesh	8962	95
3	Hypercube	8962	95

Overall delay and Network delay for Ocean application are shown in Table 5.6.

Table 5.6: Overall delay/Network delay - Ocean (4 clusters)

No	*Network used*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	9934	106
2	Mesh	9934	106
3	Hypercube	9934	106

Overall delay and Network delay for FFT application are shown in Table 5.7.

Table 5.7: Overall delay/Network delay - FFT (4 clusters)

No	*Network used*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	11392	122
2	Mesh	11392	122
3	Hypercube	11392	122

Overall delay and Network delay for LU application are shown in Table 5.8.

Table 5.8: Overall delay/Network delay - LU (4 clusters)

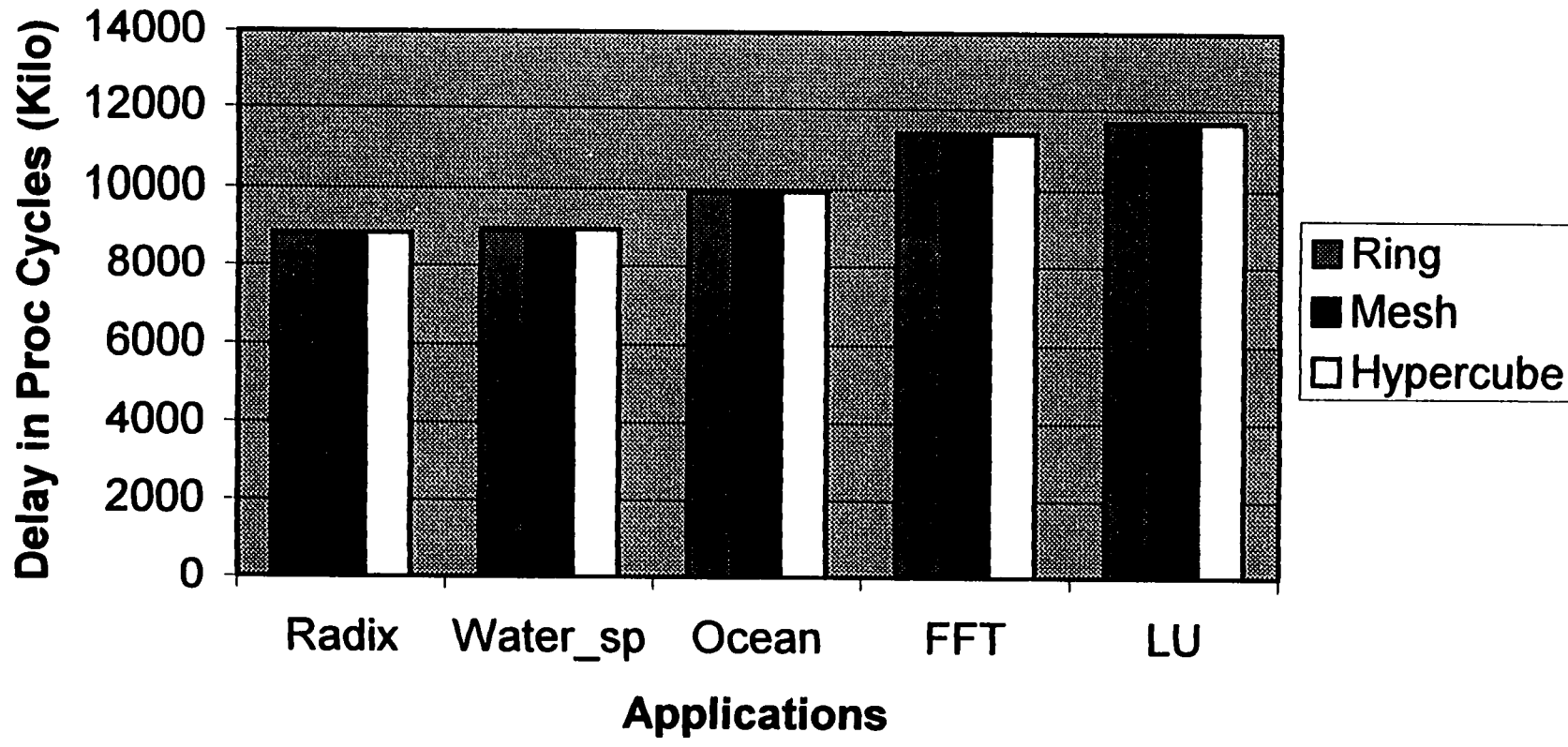
No	*Network used*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	11673	134
2	Mesh	11673	134
3	Hypercube	11673	134

Observing Tables 5.4 to 5.8, it is clear that for any application, ring, mesh, and hypercube topology always performs in the same way for a 4-cluster multiprocessor system. Overall memory latency needed for any application using any of these three networks are same as shown in Table 5.9 and Figure 5.1. Considering Figure 3.2a (page 49) and Figure 3.3a (page 52), it is clear that proposed ring, mesh, and hypercube networks are the same in architecture and they all needs the same memory delay. In Figure 3.2a node 2 can communicate with node 0 either via node 3 using inner ring, or via node 1 using outer ring, but in both cases masses passes through one intermediate node and two links. On the other hand, in Figure 3.3a node 2 can communicate with node 0 only via node 3, and the massage passes through one intermediate node and two links.

Table 5.9: Overall memory latency for a multiprocessor system with 4 clusters

Applications	*Ring delay PC in K*	*Mesh delay PC in K*	*Hypercube delay PC in K*
Radix	8849	8849	8849
Water_sp	8962	8962	8962
Ocean	9934	9934	9934
FFT	11392	11392	11392
LU	11673	11673	11673

Figure 5.1: Applications Vs Delay needed (4-cluster network)



5.3 System with 8 Clusters

A 16-processor and 8-cluster multiprocessor system, (2 processors in a cluster), Radix application needs the smallest overall network latency, and LU needs the largest overall and network latency as shown in Table 5.10 when clusters are connected through a ring network. These results are collected for a system with cache level 1 8 Kbytes and cache level2 256 Kbytes.

Table 5.10: Overall delay/Network delay - Ring Network (8 clusters)

No	*Applications (trace files)*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Radix	8968	242
2	Water_sp	9056	257
3	Ocean	10244	338
4	FFT	12028	457
5	LU	12386	473

If ring is replaced by a mesh network, total amount of delay decreases. For this mesh network Radix application needs the lowest overall and network delay, and LU needs the highest overall and network delay as shown in Table 5.11. From Figure 3.3b (page 53), message from node 0 to node 4 passes through only one link. On the other hand, message from node 0 to node 4 passes through 3 intermediate nodes (node 1, 2, and 3 or node 7, 6, and 5) and 4 links as shown in Figure 3.2b (page 50).

Table 5.11: Overall delay/Network delay - Mesh Network (8 clusters)

No	*Application (trace files)*	*Overall delay (proc clks - in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Radix	8888	162
2	Water_sp	8969	170
3	Ocean	10158	251
4	FFT	11943	372
5	LU	12298	385

If ring/mesh is replaced by a hypercube network, again Radix application needs the lowest overall and network delay, and LU needs the highest overall and network delay. Total amount of delay is the least when compared with that of a ring or mesh network. as shown in Table 5.12. From Figure 3.3b and 3.4, it is clear that most of the operations needs less intermediate nodes and/or links to be performed because of the architecture of the hypercube.

Table 5.12: Overall delay/Network delay - Hypercube Network (8 clusters)

No	*Application (trace files)*	*Overall delay (proc clks - in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Radix	8874	148
2	Water_sp	8953	154
3	Ocean	10125	217
4	FFT	11884	313
5	LU	12250	337

Overall delay and Network delay for Radix application are shown in Table 5.13 for a system with 8 clusters. Radix has the highest cache level 1 read-hit and write-hit, as a result, it takes the lowest amount of overall latency.

Table 5.13: Overall delay/Network delay - Radix (8 clusters)

No	*Network used*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	8968	242
2	Mesh	8888	162
3	Hypercube	8874	148

Overall delay and Network delay for Water_sp application are shown in Table 5.14 for a system with 8 clusters.

Table 5.14: Overall delay/Network delay - Water_sp (8 clusters)

No	*Network used*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	9056	257
2	Mesh	8969	170
3	Hypercube	8953	154

Overall delay and Network delay for Ocean application are shown in Table 5.15 for a system with 8 clusters.

Table 5.15: Overall delay/Network delay - Ocean (8 clusters)

No	*Network used*	*Overall delay (proc clks - in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	10244	338
2	Mesh	10158	251
3	Hypercube	10125	217

Overall delay and Network delay for FFT application are shown in Table 5.16 for a system with 8 clusters.

Table 5.16: Overall delay/Network delay - FFT (8 clusters)

No	*Network used*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	12028	457
2	Mesh	11943	372
3	Hypercube	11884	313

Overall delay and Network delay for LU application are shown in Table 5.17 for a system with 8 clusters.

Table 5.17: Overall delay/Network delay - LU (8 clusters)

No	*Network used*	*Overall delay (proc clks – in Kilo)*	*Network delay (proc clks - in Kilo)*
1	Ring	12386	473
2	Mesh	12298	385
3	Hypercube	12250	337

Observing Tables 5.13 to 5.17, it is clear that for any application hypercube topology is always faster, then mesh, and then ring, i.e., hypercube needs less delay than mesh and/or ring.

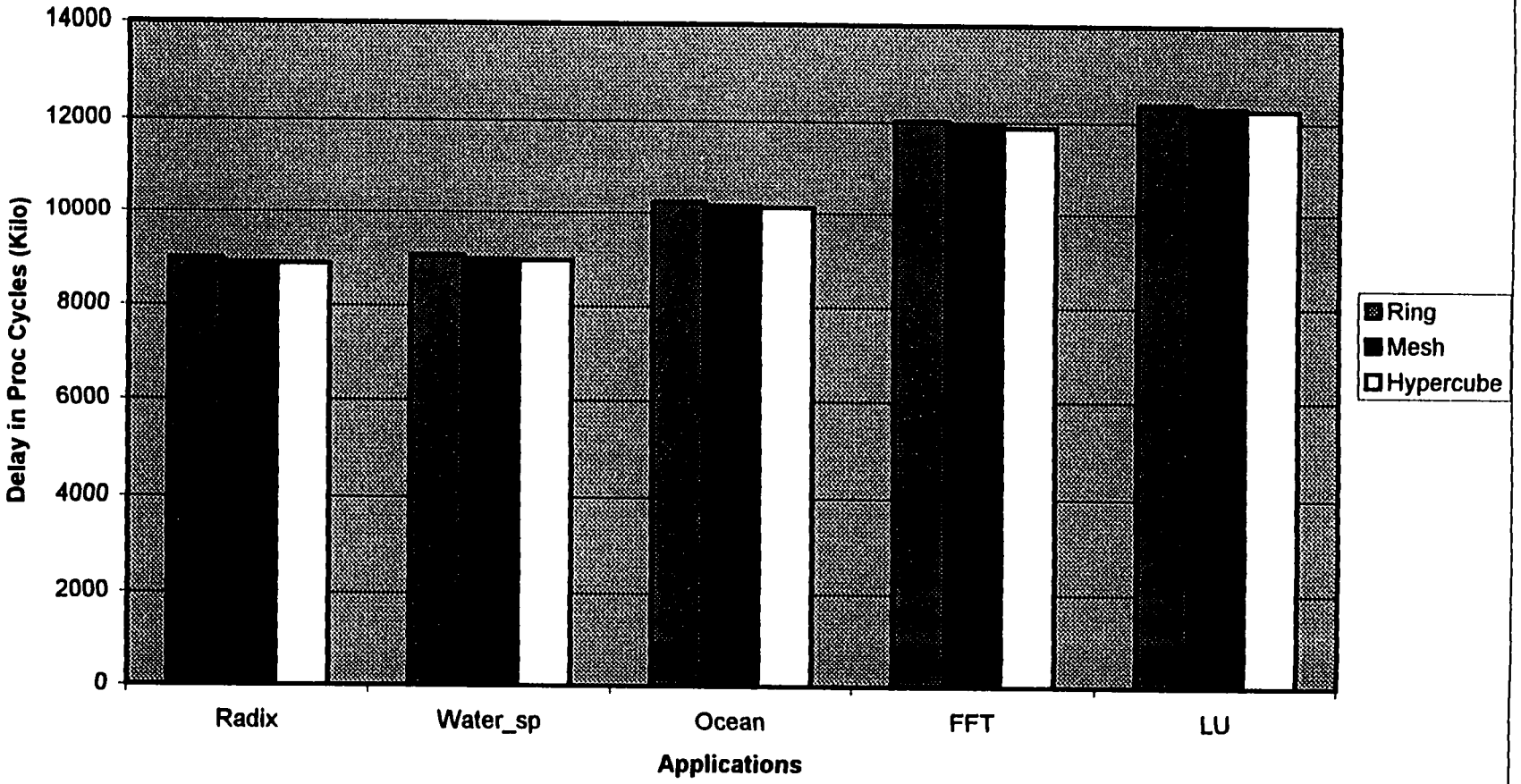
Finally, when overall memory latency needed for each specific network and application are compared, it gives very interesting results. As shown in Table 5.18 and Figure 5.2, for any application ring needs the highest amount of overall memory latency, followed by mesh and hypercube.

Table 5.18: Overall memory latency for a multiprocessor system with 8 clusters

Applications	*Ring delay PC in K*	*Mesh delay PC in K*	*Hypercube delay PC in K*
Radix	8968	8888	8874
Water_sp	9056	8969	8953
Ocean	10244	10158	10125
FFT	12028	11943	11884
LU	12386	12298	12250

Application Radix takes the lowest overall delay, followed by Water_sp, Ocean, FFT, and LU for any interconnection network.

Figure 5.2: Applications Vs Delay needed (8-cluster networks)



In Fig 5.1 all three bars for each application have the same height, indicating that ring, mesh, and hypercube networks needs same memory latency for a 4-cluster system. In Figure 5.2 three bars have different heights, for hypercube it is the smallest, for ring it the highest, and for mesh it is in between. Application Radix has the maximum cache level 1 read-hit and write-hit. So, for application Radix most of the operations are satisfied from cache level 1 without using snooping inside the cluster and the network to access the remote resource. As a result, Radix application needs minimum overall memory latency. Among the SPLASH-2 applications used in this simulation, LU has the minimum cache level 1 read-hit and write-hit. So, LU application needs the maximum overall memory latency as shown in Figure 5.2.

5.4 Impact of Cache Sizes on Latency

In this section a discussion has been made to see the impact of cache sizes on overall memory latency using a ring network. Cache Level 1 size has been varied from 8 KB to 16 KB, and Cache Level 2 size has been varied from 256 KB to 2048 KB. Table 5.19a, 5.19b, 5.20a, and 5.20b show total delay decreases with the increase of Cache Level 1 and/or Cache Level 2 size. At first results were collected for Cache Level 1 size fixed to 8 KB but Cache Level 2 size varied, then Cache Level 1 size fixed to 16 KB but Cache Level 2 size varied.

Memory latency for different applications is shown in Table 5.19 and Figure 5.3 for a 4-cluster ring network. Cache Level 1 size is 8 KB and Cache Level 2 size is 256 KB, CL1 is 8 KB and CL2 is 2048 KB, CL1 is 16 KB and CL2 is 256 KB, and CL1 is 16KB and CL2 is 2048 KB is used.

Table 5.19: Memory latency of 4 Cluster Ring Topology with variable cache sizes

Applications	*CL1 8KB CL2 256 KB PC in K*	*CL1 8 KB CL2 2048 KB PC in K*	*CL1 16 KB CL2 256 KB PC in K*	*CL1 16 KB CL2 2048 KB PC in K*
Radix	8849	8732	8764	8658
Water_sp	8962	8836	8870	8773
Ocean	9934	9811	9848	9726
FFT	11392	11273	11317	11147
LU	11673	11483	11528	11275

Mesh, hypercube, and ring networks give the same performance when used with a 4-cluster multiprocessor system. But for 8-cluster system the performance depends on the cache sizes as shown in Table 5.20a and Figure 5.4.

Table 5.20a: Memory latency of 8 Cluster Ring Topology with variable cache sizes

Applications	*CL1 8KB CL2 256 KB PC in K*	*CL1 8 KB CL2 2048 KB PC in K*	*CL1 16 KB CL2 256 KB PC in K*	*CL1 16 KB CL2 2048 KB PC in K*
Radix	8968	8926	8943	8917
Water_sp	9056	9017	9008	8995
Ocean	10244	10177	9843	9843
FFT	12028	11917	11097	11096
LU	12386	12328	12351	12314

Figure 5.3: Latency and cache sizes (4-cluster ring)

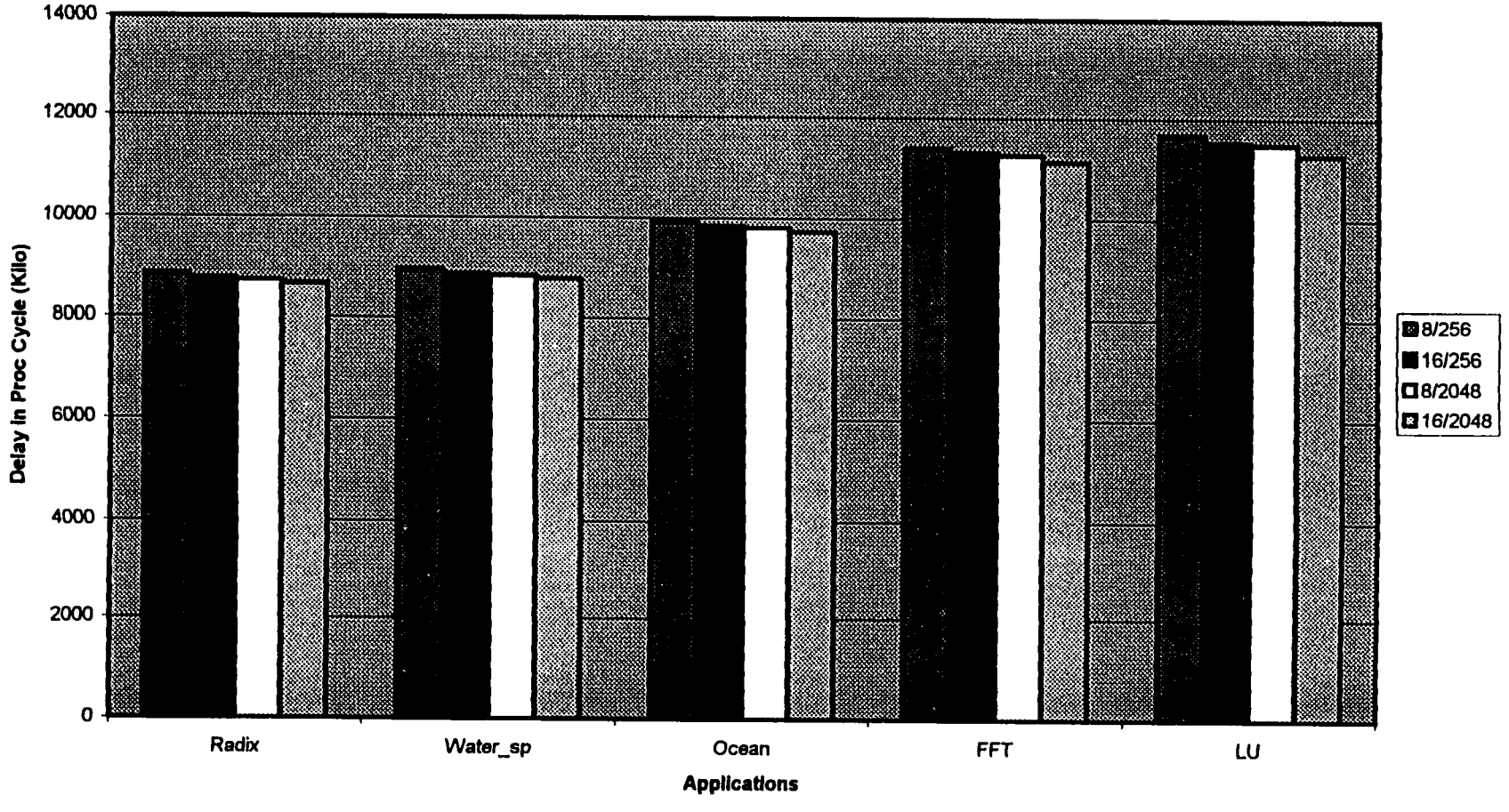


Figure 5.4: Latency and cache sizes (8-cluster ring)

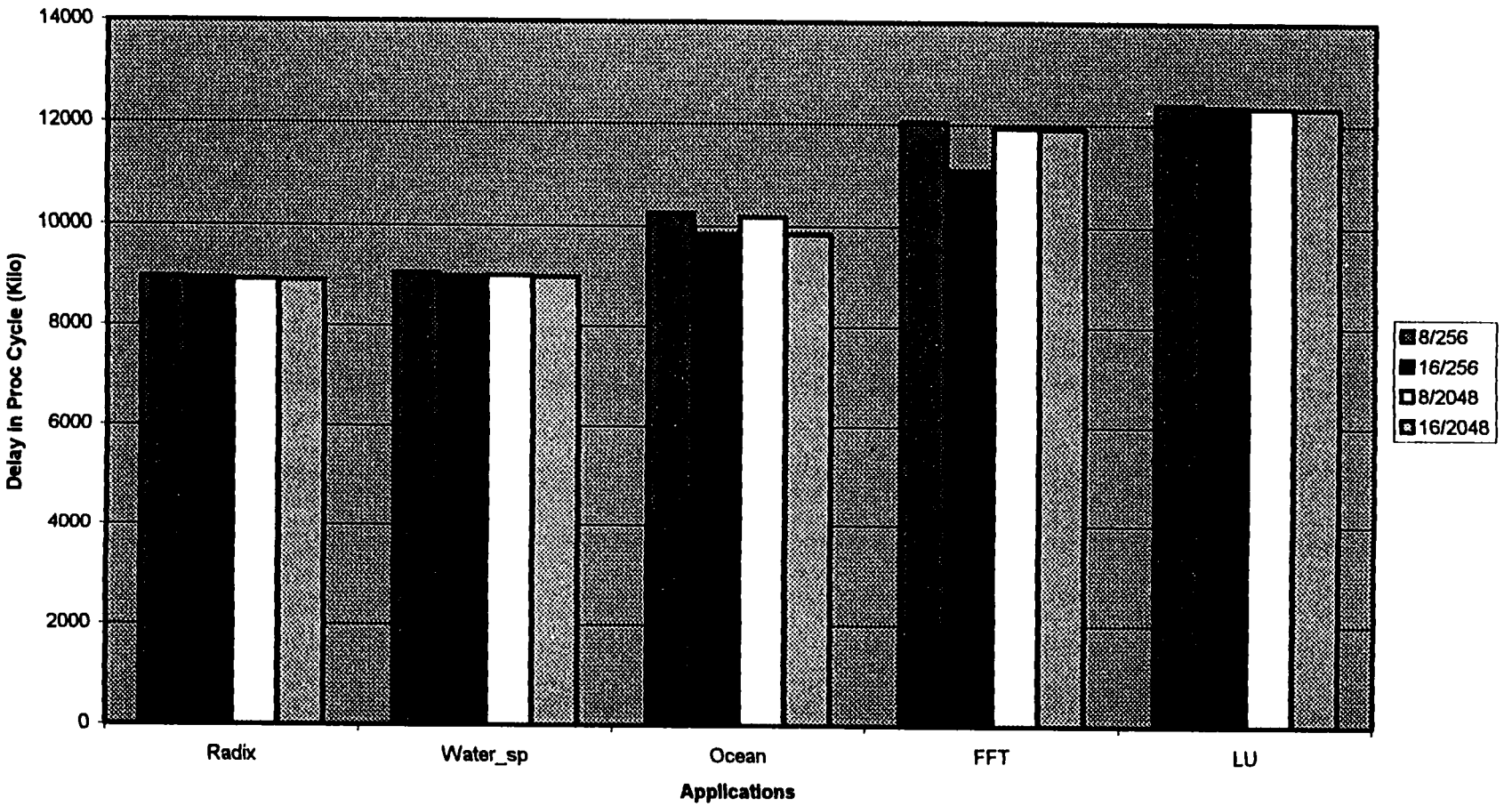


Table 5.20b: Memory latency of 8 Cluster Mesh Topology with variable cache sizes

Applications	*CL1 8KB CL2 256 KB PC in K*	*CL1 8 KB CL2 2048 KB PC in K*	*CL1 16 KB CL2 256 KB PC in K*	*CL1 16 KB CL2 2048 KB PC in K*
Radix	8888	8857	8870	8849
Water_sp	8969	8942	8935	8922
Ocean	10158	10111	9779	9779
FFT	11943	11866	11045	11045
LU	12298	12256	12296	12235

When cache sizes increase the memory latency decreases for ring, mesh, and hypercube network as shown in Figure 5.20b and Figure 5.20c.

Table 5.20c: Memory latency of 8 Cluster Hypercube Topology with variable cache sizes

Applications	*CL1 8KB CL2 256 KB PC in K*	*CL1 8 KB CL2 2048 KB PC in K*	*CL1 16 KB CL2 256 KB PC in K*	*CL1 16 KB CL2 2048 KB PC in K*
Radix	8874	8853	8862	8849
Water_sp	8953	8938	8943	8925
Ocean	10125	10087	10057	10055
FFT	11884	11811	10992	10992
LU	12250	12219	12243	12185

Figure 5.5 and 5.6 show the impact of cache sizes on overall memory latency for mesh and hypercube networks, respectively. The results say for almost all applications memory latency decreases with the increase of cache sizes. For some specific cases, this changes may not be significant. For example, delay when Ocean application is run using mesh (or ring) network remains the same even though cache level 2 is increased from 256 KB to 2048 KB (cache level 1 is fixed at 16 KB). For FFT application similar thing happens with hypercube network. This reminds the term *data pollution point* after which point delay may increase when cache sizes increases.

5.5 Summary

From Figure 5.1 and 5.2 it is obvious that hypercube network performs better than mesh and/or ring. For 4 cluster systems the hypercube behaves as a mesh, because the networks have a 2-dimensional shape. As a result the network architecture for hypercube and mesh are similar and they perform exactly the same way. Also for 4-cluster multiprocessors there is no difference between ring and mesh. So ring, mesh, and hypercube - all three networks need the same delay. But for 8-cluster systems hypercube networks get 3-dimensional shape, whereas mesh networks are 2-dimensional. And there are different routing paths between different nodes. In hypercube nodes can communicate with each other relatively faster when compared with mesh. Figure 5.2 shows the performance of hypercube is always better than mesh and ring. Routing in hypercube network becomes very faster when compared with mesh and/or ring.

Five different applications are used: Radix, Water_sp, Ocean, FFT, and LU. Overall memory latency depends on the interconnection networks as well as applications. According to the simulation results hypercube is the most efficient network followed by

Figure 5.5: Latency and cache sizes (8-cluster mesh)

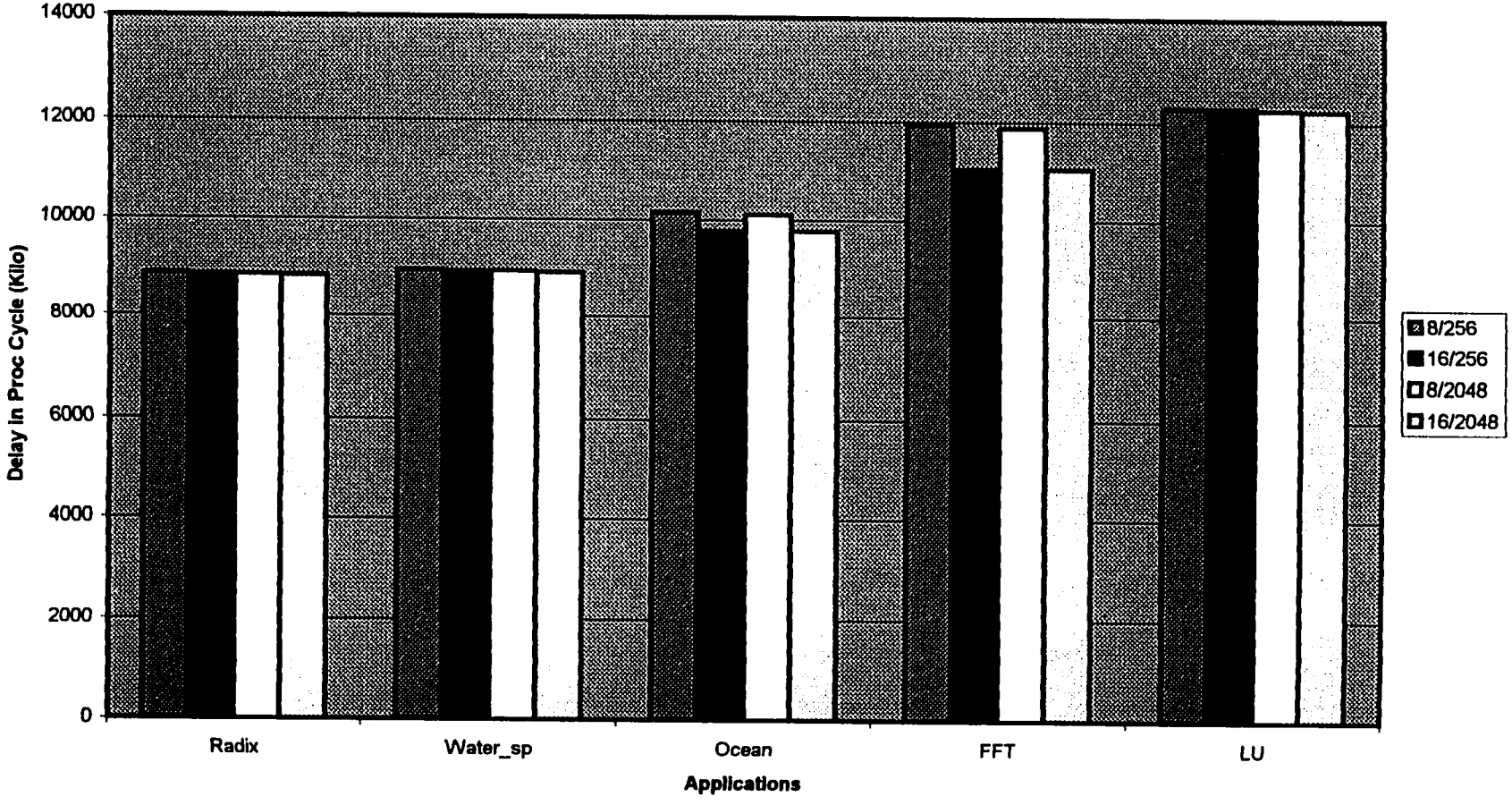
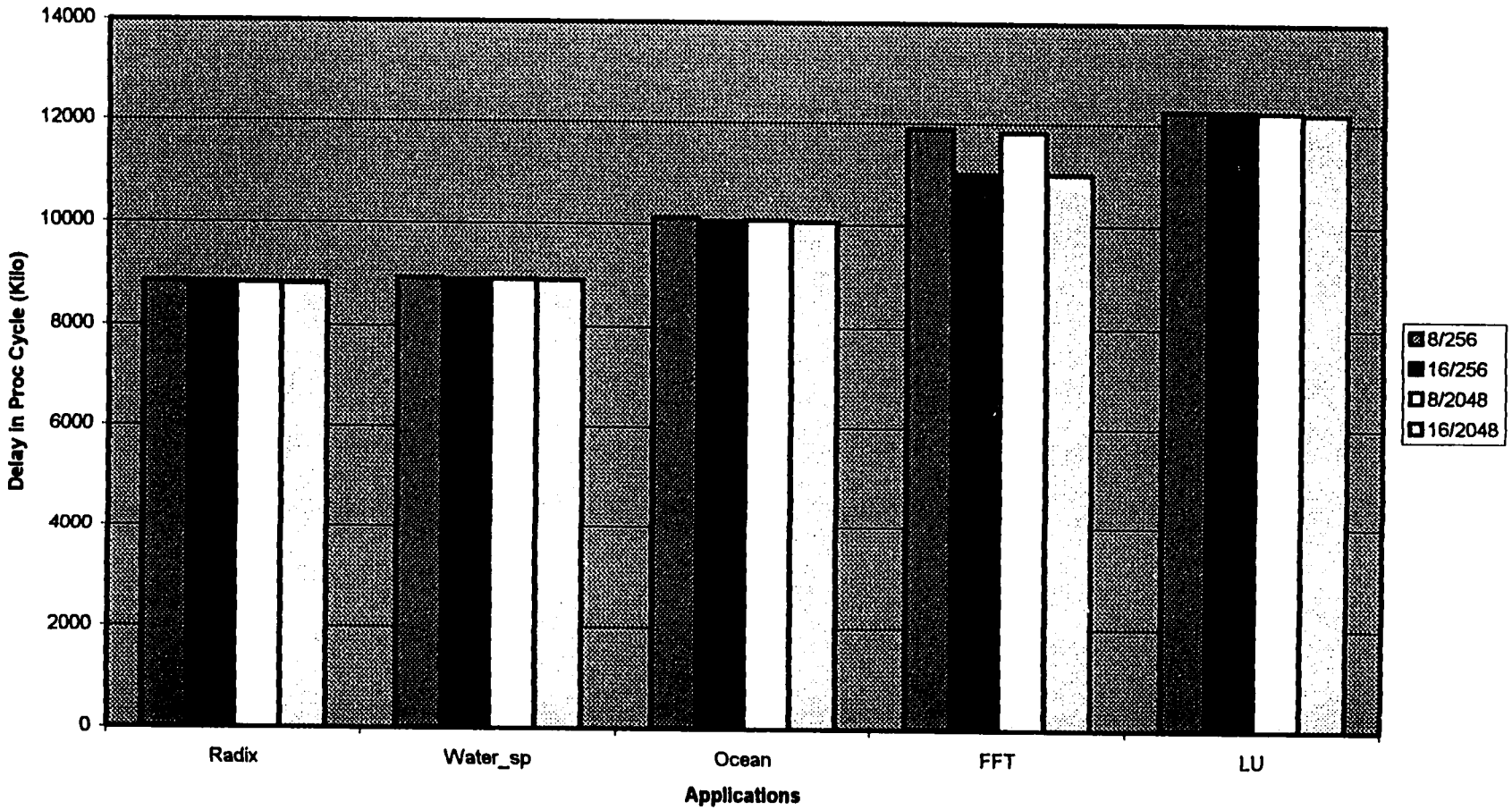


Figure 5.6: Latency and cache sizes (8-cluster hypercube)



mesh and ring network, because routing in hypercube is faster when compared to the other networks. Application Radix has the maximum (and LU has the minimum) cache level 1 read-hit and write-hit. For any network Radix application takes smallest amount of time, then Water_sp, Ocean, FFT, and LU respectively. Cache sizes also influence the performance

In this simulation memory block size has been kept constant and sizes for first level and second level caches were changed to observe the performance. For almost all applications delay decreased when cache sizes increases. For some specific cases delay does not decreases significantly. It is also remarkable that for any application total delay increases with the increase of number of clusters of the system.

Chapter 6

CONCLUSIONS AND FUTURE DIRECTIONS

In this research work, we evaluated a cluster-based cache coherence protocol for shared-memory multiprocessor system with two levels of processor caches to investigate the overall memory latency for different network topologies. Due to the availability of powerful microprocessors at low cost as well as significant advances in communication technology, a multiprocessor system is, probably, the best choice to satisfy the growing requirement for computing speed, system reliability, and cost-effectiveness. The overall performance of such a system heavily depends on the interconnection network and the type of application used.

It is almost essential for a shared-memory multiprocessor system to use processor caches to improve system performance by reducing average memory access time. Local modification inside any cache in such a system leads to cache coherence (or data inconsistency) problem, which has been the subject of extensive study for over a decade now. Both hardware-based and software-assisted solutions have been developed; these

read-operations always access the most recent version of required data item, and write-operations either update only the required memory block and invalidate all other copies, if any (called write-invalidate), or updates all existing copies (called write-update). So, the architecture for such a system should be such that message/data can be transferred among processor caches very easily.

In this research work, we simulated different network topologies in our proposed system architecture with 16 processors. We investigated the overall memory latency, using five different SPLASH-2 applications, namely, Radix, Water_sp, Ocean, FFT, and LU, for ring, mesh, and hypercube network topologies. We considered a double ring network, where nodes are connected to both rings. To communicate with each other shortest path strategy is used. A double connected mesh network is used in our simulation, where any two consecutive clusters were connected through two links. X-Y routing strategy was followed. Finally, we simulated a double connected hypercube network, where X-Y-Z routing was used. For mesh and hypercube network the same link was used to transmit a request and its acknowledgment [20][21].

We evaluated two multiprocessor systems. One with 16 processors, 4 clusters, and 4 processors in each cluster, and the other with 16 processors, 8 clusters, and 2 processors in each cluster. Inside each cluster, processors are connected by bus-based network and clusters are connected by either ring, mesh, or hypercube network. Memory latency was evaluated using proposed ring, mesh, and hypercube topologies for different sizes of cache level 1 and cache level 2.

It was observed that the cluster-based multiprocessor system with hypercube network topology outperformed those with ring and mesh topologies for all applications used. It was also observed that the overall memory latency decreased when the cache sizes were increased from 8KB to 16 KB for cache level 1 and from 256 KB to 2048 KB for cache level 2.

Future Directions

The future directions of this work include:

• Memory latency evaluation of larger multiprocessor systems with more than 16 processors.

• Simulate the cluster-based multiprocessor system with Scaleable Coherent Interface (SCI) specification the one used in the STiNG project.

BIBLIOGRAPHY

[1] Jie Wu; Distributed System Design; Florida Atlantic University, Fall 1996

[2] Per Stenstrom; *"A Survey of Cache Coherence Schemes for Multiprocessors"*; June 1990, IEEE Computer, Page 12 - Page 24

[3] Mazin S. Yousif, M.J. Thazhuthaveetil, and C.R. Das (The Pennsylvania State University); *"Cache Coherence in Multiprocessors: A Survey"*

[4] Daniel Lenoski, James Laudon, Kourosh Gharachorloo, Wolf-Dietrich Weber, Anoop Gupta, John Hennessy, Mark Horowitz, and Monica S. Lam; *"The Stanford Dash Multiprocessor"*; March 1992, IEEE Computer, Page 63 - Page 79

[5] Daniel Lenoski, James Laudon, Truman Joe, David Nakahira, Luis Stevens, Anoop Gupta, and John Hennessy; *"The DASH Prototype: Logic Overhead and Performance"*; January 1993, IEEE Transactions on Parallel and Distributed Systems, Volume 4, number 1, Page 1 - Page 31

[6] David J. Lilja; *"Cache Coherence In Large-Scale Shared-Memory Multiprocessors: Issues and Comparisons"*; September 1993, ACM Computin Surveys, Vol. 25, No. 3, Page 303 - Page 338

[7] Milo Tomasevic (Pupin Institute) and Veljko Milutinovic; *"A Survey of Hardware Solutions for Maintenance of Cache Coherence in Shared Memory Multiprocessors"*; 1993, IEEE, Page 863 - Page 872

[8] Thomas J. LeBlanc and Michael L. Scott; *"Locality Management in Large-Scale Multiprocessors"* (home page: *http:/www.cs.rochester.edu/u/leblanc/locality.html*) and Jack E. Veenstra and Robert J. Fowler (The University of Rochester); *"MINT Tutorial and User Manual":* (*ftp://ftp.cs.rochester.edu/pub/packages/mint/mint.user.manual.ps.Z*)

[9] Steven Cameron Woo, Moriyoshi Ohara, Evan Torric, and Anoop Gupta and Jaswinder Pal Singh; *"The SPLASH-2 Programs: Characterization and Methodological Considerations"*; Sep. 15, 1995, IBM 6N7A B205 EE155, 1995 ACM 0-89791-698-0/95/0006, Page 24 - Page36. Home page for the Stanford ParalleL Applications for SHared Memory (SPLASH): *http://www-flash.stanford.edu/apps/SPLASH/*

[10] Wen-Hann Wang, Jean-Loup Baer, and Henry M. Levy; *"Organization and Performance of a Two-Level Virtual-Real Cache Hierarchy"*; Proc. 16th ISCA, 1989, Page 140 - Page 148

[11] Yeong-Chang Maa and Dhiraj K. Pradhan and Dominique Thiebaut; *"Two Economical Directory Schemes for Large-Scale Cache Coherent Multiprocessors"*; September 1991, Computer Architecture News, Volume 19, Number 5, Page 10 - Page 18

[12] Jean-Loup Baer and Wen-Hann Wang; *"On the Inclusion Properties for Multi Level cache Hierarchies"*; Proc. 15th ISCA, 1988, Page 73 - Page 80

[13] Young Huang; *"A Cache Coherence Protocol for MIN-Based Shared-Memory Multiprocessors with Two Levels of Switch Caches"*; August 1995, MS Thesis

[14] Xiaola Lin, Philip K. McKinley, Lionel M. Ni; *"Deadlock-Free Multicast Wormhole Routing in 2-D Mesh Multicomputers"*; 1994, IEEE, Page 793 - Page 804

[15] Suresh Chittor and Richard Enbody; *"Performance Evaluation of Mesh Connected Wormhole-Routed Networks for Interprocessor Communication in Multicomputers"*; CH2916-5/90/0000/0647 IEEE, Page 647 - Page 656

[16] Chistopher J. Glass and Lionel M. Ni; *"The Turn Model for Adaptive Routing"*; 1992 ACM 0-89791-509-7/92/0005/0278, Page 278 - Page 287

[17] William J. Dally and Charles L. Seitz; *"Deadlock-Free Message Routing in Multiprocessor Interconnection Networks"*; 0018-9340/87/0500-0547, 1987, IEEE; Page 547 - Page 553

[18] Joon-Ho Ha and Timothy Mark Pinkston; *"A Hybrid Cache Coherence Protocol for a Decoupled Multi-Channel Optical Network: Speed Dmon"*; 0190-3918/96, 1996 IEEE, Page I-164 - Page I-274

[19] Anoop Gupta, Wolf-Dietrich Weber, and Todd Mowry; *"Reducing Memory and Traffic Requirements for Scalable Directory-Based Cache Coherence Schemes"*; The Pennsylvania State University Press, 1990, Page I.312 - Page I.321

[20] David Chaiken, John Kubiatowicz, and Anant Agarwal; *"LimitLESS Directories: A Scalable Cache Coherence Scheme"*; Proceedings of the Fourth ASPLOS, 1991, Page 224 - Page 234

[21] Per Stenstrom, Truman Joe, and Anoop Gupta; *"Comparative Performance Evaluation of Cache-Coherence NUMA and COMA Architectures"*; 1992 ACM 0-89791-509-7/92/0005/0080, Page 80 - Page 91

[22] Tom Lovett and Russell Clapp; *"STiNG: A CC-NUMA Computer System for the Commercial Marketplace"*; Nov. 14, 1996, IBM 6N7A B205 EE155, 1996 ACM 0-89791-786-3/96/0005, Page 308 - Page317