

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

RICE UNIVERSITY

**The Effects of Interconnection Networks on the
Performance of Shared-Memory Multiprocessors**

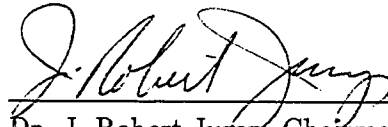
by

Usha Rajagopalan

A THESIS SUBMITTED
IN PARTIAL FULFILLMENT OF THE
REQUIREMENTS FOR THE DEGREE

Master of Science

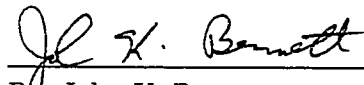
APPROVED, THESIS COMMITTEE:



Dr. J. Robert Jump, Chairman

Professor

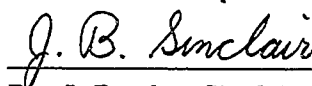
Electrical and Computer Engineering



Dr. John K. Bennett

Associate Professor

Electrical and Computer Engineering



Dr. J. Bartlett Sinclair

Associate Professor

Electrical and Computer Engineering

Houston, Texas

December, 1994

UMI Number: 1377049

UMI Microform 1377049

Copyright 1996, by UMI Company. All rights reserved.

This microform edition is protected against unauthorized
copying under Title 17, United States Code.

UMI

300 North Zeeb Road
Ann Arbor, MI 48103

ABSTRACT

The Effects of Interconnection Networks on the Performance of Shared-Memory Multiprocessors

by

Usha Rajagopalan

This thesis presents the results of a study of the effect of various interconnection network parameters on the performance of applications running on a scalable shared-memory multiprocessor. We developed a modular simulator for shared-memory multiprocessors called MEMSIM. This simulator, which was developed as a part of the Rice Parallel Processing Testbed, was used in all the experiments described in this thesis. The architecture simulated was a shared-memory multiprocessor with 64 processing nodes, with full bit-map directory-based coherence protocol. The performance of four network topologies: mesh, hypercube, and two shuffle-exchange networks were compared in our experiments. Four applications were used in our experiments: Fast Fourier Transform, Bimerge, Matrix Multiply and Successive Over Relaxation.

The main results of our study can be summarized as follows:

- With constant bisection width, the mesh network outperforms all the other network topologies
- Cache miss rate largely influences the relative performance of different network configurations

Acknowledgments

I would like to express my gratitude to Dr. Jump for his guidance in defining my goals and keeping me on track to achieve them. I would like to thank Dr. Sinclair and Dr. Bennett for their guidance this past year during Dr. Jump's leave of absence. I would also like to thank all three of them for serving on my thesis committee and for their patient review of my thesis draft.

I would like to thank Sandhya for her help with the cache simulator and all her suggestions that improved this work. I would like to thank Ram and Sridhar for their help and suggestions during our weekly meeting. I would like to thank all my colleagues for making my experience at Rice very memorable.

Last but not the least, I would like to thank my family for their support from near and far in completing this thesis.

Contents

Abstract	ii
Acknowledgments	iv
List of Illustrations	viii
List of Tables	xi
1 Introduction	1
2 Overview	3
2.1 Overview of Interconnection Networks	3
2.1.1 Overview of Interconnection Network Topologies	3
2.1.2 Overview of Interconnection Network Switching Mechanisms .	4
2.1.3 Overview of Interconnection Network Routing Algorithms . .	5
2.2 Motivation	5
2.3 Related Work	7
3 Simulation Environment	10
3.1 Implementation of MEMSIM	12
3.2 Processor Module	14
3.3 Profiling	14
3.4 Cache Module	16
3.5 Write Buffer Module	17
3.6 Bus Module	18
3.7 Memory Module	19

3.8	Directory Module	20
3.9	Network Interface	21
3.10	NETSIM	21
4	Experimental Setup	22
4.1	Processor	22
4.2	Cache	24
4.3	Memory	24
4.4	Directory	25
4.5	Interconnection Networks	25
4.6	Coherence Protocol	28
5	Results	32
5.1	Benchmarks	32
5.1.1	Matrix Multiplication (MMULT)	33
5.1.2	Successive Over Relaxation (SOR)	33
5.1.3	Fast Fourier Transform (FFT)	34
5.1.4	Sorting (Bimerge)	36
5.2	Effect of Network Topology	37
5.2.1	Simulation of Networks with Constant Channel Width	37
5.2.2	Simulation of Networks with Constant Bisection Width	50
5.3	Components of Latency of a Cache Line Fetch	54
5.4	Effect of Memory Speeds on Performance	63
5.5	NETSIM Simulation Complexity	66
6	Conclusions	75
6.1	Conclusions on Network Topology Experiments	76
6.2	Conclusions on Comparing Different Link Widths	77

6.3	Conclusions on Network Complexity	78
6.4	Future Work	79
	Bibliography	81

Illustrations

3.1	Routines used to specify architectures in MEMSIM	13
4.1	Scalable Shared-Memory Multiprocessor	23
4.2	State Diagram of the Write-Back Coherence Protocol	29
4.3	State Diagram Showing Different States of a Directory Line	30
5.1	Communication Pattern of the FFT Algorithm	35
5.2	Performance of Different Network Topologies for FFT	38
5.3	Performance of Different Network Topologies for Bimerge	39
5.4	Performance of Different Network Topologies for SOR	39
5.5	Performance of Different Network Topologies for MMULT	40
5.6	Communication Pattern of SOR on a Mesh Network	45
5.7	Communication Pattern of SOR on a Hypercube Network	45
5.8	Communication Pattern of MMULT on a Mesh Network	47
5.9	Communication Pattern of MMULT on a Hypercube Network	47
5.10	Performance of Different Network Topologies for Inefficient MMULT .	49
5.11	Performance of Different Network Topologies with Constant Bisection Width for FFT	51
5.12	Performance of Different Network Topologies with Constant Bisection Width for Bimerge	52

5.13 Performance of Different Network Topologies with Constant Bisection Width for SOR	52
5.14 Performance of Different Network Topologies with Constant Bisection Width for MMULT	53
5.15 Sum of Average Network Latency For Request and Reply in a Mesh Network	56
5.16 Sum of Average Network Latency and Port Waiting Time For Request and Reply in a Mesh Network	57
5.17 Average Service Time and Waiting Time at Directory and Memory Module in a Mesh Network	60
5.18 Overall Cache Miss Access Latency in a Mesh Network	60
5.19 Performance Difference in FFT as the Memory Speeds are Increased .	64
5.20 Performance Difference in SOR as the Memory Speeds are Increased .	65
5.21 Execution Time Predicted by Detailed Network Simulation and Approximate Network Simulation for FFT	67
5.22 Execution Time Predicted by Detailed Network Simulation and Approximate Network Simulation for Bimerge	68
5.23 Execution Time Predicted by Detailed Network Simulation and Approximate Network Simulation for SOR	69
5.24 Execution Time Predicted by Detailed Network Simulation and Approximate Network Simulation for MMULT	70
5.25 Simulation Time Taken by Detailed Network Simulation and Approximate Network Simulation for FFT	71
5.26 Simulation Time Taken by Detailed Network Simulation and Approximate Network Simulation for Bimerge	72
5.27 Simulation Time Taken by Detailed Network Simulation and Approximate Network Simulation for SOR	73

5.28 Simulation Time Taken by Detailed Network Simulation and Approximate Network Simulation for MMULT	74
---	----

Tables

5.1	Applications Used in Study	36
5.2	Comparing the Performance of Shared-Memory Multiprocessors with Different Network Topologies	38
5.3	Equivalent Link Widths	51
5.4	Latency of the Two Packet Sizes at Various Link Widths	56
5.5	Percentage Improvement in Cache Miss Latency with a Mesh Network When Link Width is Increased	61
5.6	Percentage Improvement in Performance with a Mesh Network When Link Width is Increased	61
5.7	Percentage Improvement in Performance as the Link Widths Are Increased. (2 Memory Speeds)	66

Chapter 1

Introduction

This thesis studies the effect of various interconnection networks and interconnection network parameters on the performance of applications running on a scalable shared-memory multiprocessor. We developed a modular simulator for shared-memory multiprocessors called MEMSIM. This simulator, which was developed as part of the Rice Parallel Processing Testbed (RPPT), was used in all the experiments described in this thesis.

It is becoming increasingly more difficult to satisfy the demand for high performance computer systems by using uniprocessors. As a result, most new high performance computing systems are multiprocessors constructed from commercial microprocessors. Two programming models have been developed for these systems, the distributed-memory programming model and the shared-memory programming model.

Programming for a distributed-memory programming model is felt to be harder than for a shared-memory model, since the programmer has to provide explicit statements to facilitate data communication between processors. Examples of multiprocessors which support the distributed-memory programming model include the Intel Paragon [9], the Thinking Machine Corporation's CM-5 [12], and the Cray T-3D [31]. Shared-memory multiprocessors have received a lot of attention because the shared-memory programming model is felt to be easier to use. A majority of available shared-memory multiprocessors use a single-bus and are not scalable. Examples of such systems include the Sequent Symmetry [33] and the Encore Multimax [10]. The

bus, which is a central resource, becomes a bottleneck when too many high performance processors use the bus to access memory. A single-bus shared-memory system can only support about 30 processors.

Multiprocessors can be scaled to more than a few tens of processors, by using an interconnection network that can connect a few hundred or even a few thousand processors without becoming a bottleneck. Such large-scale multiprocessors can support the shared-memory programming model by using a directory-based coherence protocol [6]. A simple directory-based coherence protocol has a directory entry associated with each cache line-sized block in memory. This directory keeps track of nodes that have cached a copy of that line and the current state of the line. This information can be used to send messages to the caches involved in a specific transaction instead of broadcasting information about each transaction to all caches. The Stanford DASH multiprocessor [32] and MIT's Alewife [3] use the directory-based coherence protocol. This thesis studies the performance of interconnection networks in a large-scale shared-memory multiprocessor using a directory-based coherence protocol. This study is conducted by simulating the execution of parallel applications on such a multiprocessor.

The performance of four network topologies: mesh, hypercube, and two shuffle-exchange networks is compared. The various components of the network latency is analyzed and the factors in the application that affect these components is studied. The tradeoff between time to run a detailed network simulation and the accuracy of the results predicted by the simulation is also studied.

Chapter 2 gives some background information, an overview of the project, and related work. Chapter 3 describes the MEMSIM simulator. Chapter 4 describes the simulated architecture and the parameters used in the experiments. Chapter 5 presents and discusses our results. Chapter 6 provides conclusions and future direction of this research.

Chapter 2

Overview

This chapter provides an overview of interconnection networks, describing the different topologies, switching methods and the routing algorithms used in networks. It also describes the network types that are studied in this thesis, a motivation for studying these particular networks and the motivation for studying the effect of networks on the performance of shared-memory multiprocessors. This chapter also describes related work done in the study of interconnection networks.

2.1 Overview of Interconnection Networks

The interconnection network is a very important component in a multiprocessor. The purpose of an interconnection network is to provide communication paths between the modules of a parallel system. The different parameters to be considered during the design of an interconnection network are the network topology, the routing algorithm, switching mechanism and the bandwidth of the network.

2.1.1 Overview of Interconnection Network Topologies

The simplest network topology is a bus, a single path that all the modules must share. This is not scalable in terms of performance, since only one module can send data at a time. The crossbar network represents the other extreme where each module has a direct connection to every other module in the system. This is not scalable in terms of cost since it requires N^2 switches to connect N modules. In between these two extremes there are a variety of network topologies that one can choose from. They

are generally classified as direct networks or indirect networks. In direct networks every switch in the network is also connected to a processing node. The direct k -ary n -cube networks have n dimensions with k nodes in each dimension. The 2-D torus and the hypercube, which is a binary n -cube, represent two extremes of this class of networks. Indirect networks, also known as multistage interconnection networks, have the characteristic that all modules connected to the networks are equidistant from each other. The bisection width of a topology is the minimum number of channels that must be cut to divide the network into two equal halves. It is used as a measure of network cost, since the complexity of a connection is wire-limited. The different network topologies studied in this thesis are mesh, hypercube and the shuffle-exchange network.

2.1.2 Overview of Interconnection Network Switching Mechanisms

Switching methods include packet-switching, circuit-switching, virtual cut-through [30] and wormhole routing [14]. In packet-switching, also known as store-and-forward, the entire packet is stored in the buffer of an intermediate node before it is forwarded to the next node determined by the routing algorithm. In this case the latency of a packet to traverse the network is proportional to the product of the size of the packet and the distance (number of hops) traversed by the packet. In circuit-switching a path is first established between the source and destination nodes and this path is reserved for the packet. The tail of the packet tears down the path. This can achieve better performance than packet-switching for long packets. Virtual cut-through sends a packet through the network without first establishing a circuit. As long as there is no blocking the head of the packet makes progress through the network. The other flits (smallest unit of data that can be transferred between two switches) of the packet can be spread out along the path, depending on the length of the packet. When the packet is blocked at a node, trailing flits catch up and are buffered at that node.

The difference between this and wormhole switching is that in the latter, the switch buffers do not have to be large enough to hold a whole packet. The flits of a blocked packet are distributed along the network in different buffers. The routing algorithms are more limited for wormhole routing, due to the possibility of deadlock. Wormhole routing is being used widely in current generation multiprocessors. This thesis studies the performance of wormhole-routed networks.

2.1.3 Overview of Interconnection Network Routing Algorithms

Routing determines the path selected by a packet in order to reach its destination. Routing can be classified as deterministic or adaptive. In deterministic routing, also known as oblivious routing, the path is completely determined by the source and destination pair. Adaptive routing takes the state of the network into consideration and can alter the path of a packet to avoid congestion in the network. The routing algorithm is minimal if the path selected is a shortest path between the given source and destination pair. One deterministic routing scheme that is guaranteed deadlock-free for wormhole routing is a dimension-ordered routing scheme called *e*-cube routing [35]. Adaptive routing algorithms for wormhole-routed networks, which are deadlock-free, are complicated and can be expensive and slow to implement [35]. In this thesis, the study of networks is restricted to deterministic routing.

2.2 Motivation

Shared-memory multiprocessors that scale up to a few thousand processors seem viable. Much research is focused on making such systems efficient. In particular, efforts are made to hide or tolerate the high memory access latencies involved in such systems. One technique is to provide fast context switching in hardware. Several threads can be kept active on each processor at the same time and a processor can switch among these threads to hide individual access latency [44]. Dynamic instruction scheduling

at the processor can increase processor utilization, by allowing instructions to execute out of sequence whenever earlier instructions are waiting on a data access, provided there is no data dependency [21]. Such techniques attempt to increase processor and network utilization. Previous simulation experiments used to evaluate these techniques typically have ignored contention in the network. While this could give an estimate of the best performance gain possible, if the network performance does not match the performance of the rest of the system, these gains will be unattainable.

Interconnection networks are important components of multiprocessors. The rate of execution by a single processor is increasing, through increased clock speed, instruction pipelining and the use of multiple functional units with super-scalar execution. Without the proper design of interconnection networks the communications among parallel threads may result in significant performance degradation due to network latencies and throughput limitations.

Interconnection networks have been studied for a long time. Much attention has been given to the design and performance analysis of interconnection networks. Numerous network topologies have been proposed and evaluated. A survey can be found in [18]. The performance analysis of networks has mostly been done using mathematical models and stand-alone simulations ([16, 1, 2, 13]). However, such performance evaluation techniques do not always capture the access patterns of real programs. One of the motivating factors in this work is to study the performance of networks driven by the execution of an application on a shared-memory multiprocessor.

This thesis compares the performance of four different network topologies, the mesh, the hypercube and two shuffle-exchange network configurations. These topologies were chosen because commercial multiprocessors have been built using mesh, hypercube and multistage interconnection networks. The hypercube has been used in several multiprocessors, starting with a research machine from Caltech called the

Cosmic Cube [40]. Other multiprocessors with hypercubes include the iPSC and iPSC/2 [8] from Intel, NCUBE/ten [11], and CM-2 [25], an SIMD machine from Thinking Machines Corporation. More recently Intel has abandoned the hypercube architecture and is using a mesh network in the Paragon [9]. The Cedar parallel processor from the University of Illinois [19] uses a shuffle-exchange network with an 8x8 crossbar at each switching node.

The performance of the different network topologies are compared when they have an equal link width and when they have a constant bisection width. Also, the performance of each network topology is studied for different link widths and in order to find an optimal bandwidth for a particular network topology. For all of these experiments the other parameters of the system, such as speed of the processor and memory sub-system and size of the caches, are fixed and the network performance is studied under these fixed parameters. All the system parameters reflect current technology and the technology trends expected. The network performance is also studied at different ratios of processor speeds to memory speeds, since we do not expect DRAM performance to keep up with the improvements in processor performance. All our experiments use wormhole-routed networks since this is the routing scheme of choice in most commercial multiprocessors.

2.3 Related Work

Interconnection networks have been studied for a long time. The performance analysis of networks has mostly been done using mathematical models and stand-alone simulations ([16, 1, 2, 13]). [29] presents an algebraic theory based on tensor products for modeling direct interconnection networks. Ponnuswamy et al. evaluate the performance of the network in the CM-5 multiprocessor experimentally [37]. Chittor and Enbody evaluate the performance of wormhole routed meshes with the use of the Symult 2010 multiprocessor [7]. They use experimental results to show that the path

length of the message does not adversely affect performance, but that contention for network resources does. Neither of these studies use complete applications, but drive the network with messages of varying sizes and different network traffic patterns.

There has been a lot of work to improve a particular algorithm for a specific network topology. Examples include the implementation of FFT on hypercubes [45] and an implementation of FFT on the butterfly network [34]. Such work studies the network performance with respect to a specific algorithm and may not be useful in the design of a general-purpose multiprocessor. [4] presents the results of running the NAS benchmarks on several parallel processors and supercomputers, including the iPSC/860, which has a mesh network and the BBN TC2000, which has a butterfly interconnection network. However, no attempt is made to analyze the results and determine whether the differences are due to the network or due to other differences in the two systems. In fact, it is very difficult to separate the performance effect of networks from other factors and might require some diagnostic hardware to be installed in the machines.

Abraham and Padmanabhan compared the performance of the direct binary n -cube with the indirect binary n -cube network [1]. The performance evaluations are done for equiprobable distribution of message destinations. They also evaluated the network under special conditions such as broadcasts and hot-spots. The performance analysis was done for packet-switched networks. They found that the direct network performs better. Dally compared the performance of networks belonging to the class of direct k -ary n -cube [13]. He compared the networks for wormhole routing under a constant bisection width constraint for a network size of 256, 16 K, and 1 M nodes. He arrived at values for the actual latency and latency with contention when the traffic in the network is a certain fraction of the total network bandwidth. His results show that the 2D-torus network performs best. Agarwal expanded on this work in [2], by calculating the latency of k -ary n -cube networks when the switch latency is

greater than the link latency. He found that when the switch delay is higher, 3D and 4D tori perform better than the 2D-torus.

The simulator used in the experiments described in this thesis is an execution-driven simulation testbed called the Rice Parallel Processing Testbed (RPPT). As a part of this thesis we extended RPPT to support shared-memory simulation, by developing a cache-memory hierarchy simulator called MEMSIM. A detailed network simulator called NETSIM [27] is a part of RPPT and is used in our experiments. Tango [15], Proteus [5] and an earlier RPPT shared memory simulator [17] are other execution-driven simulators running on uniprocessors. The Wisconsin Wind Tunnel (WWT) [39] executes on a CM-5 using a form of distributed discrete-event simulation. A major difference between MEMSIM and the above mentioned simulators is that MEMSIM interfaces to a detailed and versatile interconnection network simulator (NETSIM). NETSIM can simulate different types of routing such as virtual cut-through, worm-hole routing and store-and-forward. Most interconnection networks can be built from the modules provided by NETSIM. The WWT and DASH [32] simulator (using Tango) approximate the effect of the network with constant latency to reach any destination. Proteus allows the user to simulate interconnection networks with store-and-forward routing or approximate the effects of the interconnection network using an analytical model. Store-and-forward routing is not used in newer parallel machines since cut-through routing techniques have lower latency. The WWT simulates one target processor per host processor of the CM-5, limiting the number of simulated processors to the size of the CM-5.

Chapter 3

Simulation Environment

The simulator used in the experiments described in this thesis is an execution-driven simulation testbed called the Rice Parallel Processing Testbed (RPPT). As a part of this thesis we extended RPPT to support shared-memory simulation, by developing a cache-memory hierarchy simulator called MEMSIM. A detailed network simulator called NETSIM [27] is a part of RPPT and is used in our experiments to model the interconnection networks. This chapter provides an overview of MEMSIM.

In the design of parallel machines, simulators provide an effective tool to evaluate different architectural features. Simulations driven by the execution of applications have proven to be fairly accurate and are cost-effective when compared to building hardware prototypes. Simulators that are not driven by real programs, but instead use analytical models or are distribution-driven, are not sufficiently accurate for many purposes, because they do not accurately model the behavior of real programs in this way. Different techniques used in the simulation of the execution of an application on a target architecture are instruction-level emulation, trace-driven simulation and execution-driven simulation.

Trace-driven simulations use address traces collected from a run of the program on a hardware machine. Traces are collected from a machine when an application is running in that machine. The traces are then input to a simulator modeling the target architecture. This simulation technique has been used for a long time. One of the advantages of trace-driven simulation is that one does not need to have access to the source code of the application. However, trace-driven simulation is not very accurate

in the case of multiprocessors because it is difficult to obtain accurate interleaving of address traces from different processors in the multiprocessor. Any interleaving inherent in the trace may not be valid for the execution on the target architecture. Another disadvantage of this technique is that the address traces generated for an application will generally require large amount of storage space. Instruction-level simulators emulate each instruction of the target machine in software. Such simulations can be quite expensive in terms of the time taken to simulate the target architecture.

In the execution-driven simulation approach, the machine language instructions of the application program are executed directly on the simulation host. The program is modified using a profiler that inserts instructions in every basic block of the program to increment the simulation time as the program executes. In systems with caches, the simulator has to determine whether each access is a hit or a miss in the cache, and in the case of a miss, simulate a new cache line fetch. The global clock of the simulation is updated during cache accesses and other potential processor interaction points. When the memory address can be determined statically, the profiler generates these addresses and writes them to a trace file. The profiler inserts code into the user program to generate all other addresses dynamically.

The overhead of emulating each instruction is avoided in execution-driven simulation, making it faster than instruction-level simulators. Since the execution-driven simulation executes all instructions on the host machine, the cache data structure in the simulator does not maintain a copy of the cache line. The simulator only maintains the tags and state of the simulated cache. This makes this type of simulator more space efficient than instruction-level simulators.

MEMSIM is a modular execution-driven simulator for shared memory architectures. It is implemented as an extension to a discrete-event simulator called YACSIM [26], which was developed as part of the Rice Parallel Processing Testbed (RPPT). MEMSIM can also be driven by traces. MEMSIM was used to obtain all results pre-

sented in this thesis. MEMSIM interfaces with NETSIM [27], a general-purpose inter-connection network simulator that is also implemented as an extension to YACSIM. The different modules provided by MEMSIM are cache, write buffer, bus, directory, memory, network interface and processor. The features of these modules are explained in the following sections. The user specifies a target architecture in MEMSIM by initializing the necessary modules and then connecting them to configure the architecture. These modules may be chosen from the set of modules already implemented, or may be supplied by the user.

3.1 Implementation of MEMSIM

All modules in MEMSIM are implemented using YACSIM activities. YACSIM implements two types of activities, events and processes. Activities can be scheduled to “happen” in the future. Each YACSIM activity when created is assigned a C-procedure that specifies its action. This procedure is called the *body* of the activity. A YACSIM process can temporarily suspend execution and continue execution at the next line of the process body when it is reactivated. This is achieved by saving the current context when the process is suspended and restoring the context when it is rescheduled. A YACSIM event starts executing in the beginning of the event body each time it suspends and is reactivated. If MEMSIM modules are implemented as processes, the overhead associated with suspending and reactivating the process can get quite expensive when several MEMSIM modules have to be scheduled to complete one remote memory access. The modules in MEMSIM are implemented using YACSIM events. By changing the structure of the event body to keep track of the last line executed before suspension, events can be as flexible to use as processes.

During the design and development of MEMSIM much emphasis was given to modularity. While developing such a simulator, the developer cannot foresee all the different architectures that are going to be evaluated using the simulator. Therefore

it is important that the tools are provided for the user to customize a module or implement a new module altogether and have it work with existing modules.

Each module in MEMSIM has a data structure that is a derivative of a base module type. The user sets up the architecture to be simulated, by creating the necessary number of modules and then connecting the modules together using MEMSIM calls (see Fig. 3.1). When a module is created a YACSIM event is initialized and associated with that module. The body of the event is the main function that acts on the data structure of that module in order to update it. Scheduling the event associated with a module does not have any undefined side-effects on other modules. This makes it possible for the user to connect modules as necessary or to define and use new modules connected to existing MEMSIM modules.

A request packet is created and initialized at the processor module, which is the first module entered during simulation. From there the request is sent from one module to another until it is satisfied. For example, a request would be satisfied in

```

Pcr1 = NewProcessor (parameters)
Cache1 = NewCache (parameters)
Pcr2 = NewProcessor (parameters)
Cache2 = NewCache (parameters)
Bus = NewBus (parameters)
Memory = NewMemory (parameters)

Connect (Pcr1,  port=0, Cache1, port=0)
Connect (Pcr2,  port=0, Cache2, port=0)
Connect (Cache1, port=1, Bus,    port=0)
Connect (Cache2, port=1, Bus,    port=1)
Connect (Bus,   port=2, Memory, port=0)

```

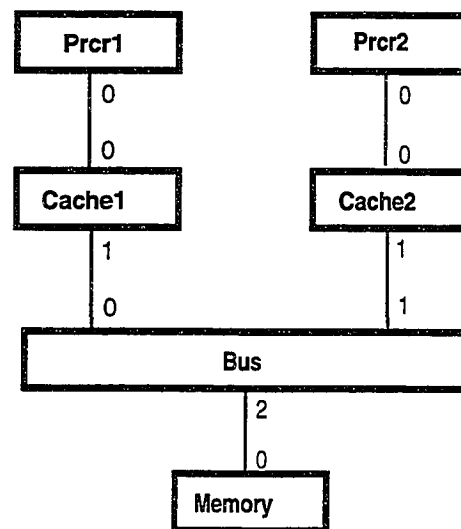


Figure 3.1: Routines used to specify architectures in MEMSIM

the cache if there is a cache hit. Otherwise, the request would have to be sent to the next level in the memory hierarchy. When a request is satisfied, it travels back to the processor and control is returned to the user program. If a module is connected to several modules, a routing function is used to determine the next module requested.

The different modules supported by MEMSIM include Processor, Cache, Write Buffer, Bus, Directory, NETSIM interface and Memory. A short description of the features of each module follows.

3.2 Processor Module

In execution-driven simulation the application program is executed directly on the machine on which the simulation runs. The application program is initially augmented with lines of code by a program called the profiler. The code added to the application program by the profiler helps the simulation to keep track of the number of cycles executed and provides hooks to simulation modules such as caches. Currently RPPT supports a profiler for the SPARC architecture when used as a shared-memory multiprocessor simulator. Therefore the processor module in MEMSIM simulates the SPARC architecture. The next section provides an overview of the MEMSIM profiler.

3.3 Profiling

The profiler works on the assembly code of the application program. In order to account for the time taken to execute the program the profiler inserts a few instructions at the end of each basic block to increment the timing variable of YACSIM by the number of cycles it takes to execute that basic block. Two versions of the profiler are available for use with MEMSIM. The user can choose to simulate both instruction and data accesses to the cache, or simulate the data accesses alone (assuming in this case that all instruction accesses hit in the cache).

In order to simulate the data accesses to the cache the profiler inserts code before each instruction that has a data access (various load and store instructions in the case of the SPARC), to do the following:

- save global registers
- extract the address for the data access and write the address to a variable
- determine the type of access
- call the cache module with this information
- restore global registers after the return from the cache module

The profiler saves the global registers into the data structure for the current process.

When instruction accesses are also simulated, the profiler determines the instruction addresses at profile time. The user provides the profiler with assembly code of the application program as well as an executable of the program. This executable is compiled by the user from the application program before the profiler augments the code. This gives the profiler an access to the accurate address trace of the program. However, this executable is only used by the profiler and the profiled application program is the one that is actually executed.

The profiler extracts the instruction addresses from the symbol table of this executable and writes the static address trace into a file. Labels are inserted into the static trace to indicate basic block boundaries. It must be mentioned here that static address traces are usually much shorter than dynamic address traces that are generated in a trace-driven simulation environment. This is because programs usually spend about ninety percent of their time looping in ten percent of the code. The instruction accesses are simulated at the beginning of each basic block for all instruction addresses in that basic block. The profiler inserts code similar to that mentioned above for the data accesses, at the beginning of each basic block. During each call,

MEMSIM simulates all instruction accesses for the basic block (by reading the address trace produced by the profiler) before returning to the user program.

3.4 Cache Module

The cache module simulates a processor cache. It can be a unified cache, a data cache, or an instruction cache. It can also be used at any level in a hierarchy of caches. The cache module is implemented as a data structure that stores the tag and state for each cache line. The user can set the size of the cache, the line size, the set associativity, the replacement policy, and the coherence protocol when the module is initialized. It supports any cache size starting from 1 KB, limited only by the amount of memory available for the simulation. The module can also be used to model an infinite cache. Set size may be any power of two, making MEMSIM capable of simulating a direct-mapped cache, a fully associative cache or any set size in between. The replacement policies supported by the cache module are Least Recently Used (LRU) and First In First Out (FIFO). The cache module can be customized to support any coherence protocol desired. When the cache module is initialized, the coherence protocol is supplied to it as a pointer to a function. The user can simulate one of the coherence protocols supported by MEMSIM or provide a function to simulate any other coherence protocol.

The coherence protocols supported by MEMSIM include Write-Through with Update, Write-Through with Invalidate, Write-Back with Invalidate and Write-Through when Shared. The Write-Through caches update the main memory each time there is a write to the cache. The Write-Through with Update protocol updates other caches that have a copy of the data when one node writes through to memory. The Write-Through with Invalidate protocol invalidates other cached copies when one node writes through to memory. In Write Back caches, on a write access to a shared line, the cache sends a request to invalidate all other cached copies and

obtain an exclusive copy. Subsequent writes to the line are completed at the cache. This cache has the only valid data in the system, and the line is in the dirty state. When the cache wants to replace the line or another processor wants to read or write this line, the data is written back to main memory. With the Write-Through when Shared protocol, if the line is in shared state, on a write access the data is written through to memory. At this time all other copies are updated. When this cache has the only copy of the line, writes are completed locally. In this case also, the line is dirty and transferred to memory later.

Caching is said to be *adaptive*, if the programmer has control over the coherence protocol. MEMSIM supports adaptive caching by allowing the user to inform the simulator that certain address ranges are associated with a particular coherence protocol. The cache type must also be initialized to be adaptive and given a default coherence protocol. The cache module checks for the coherence type of each access when it is referenced for the first time. If there is no entry for that particular address, the default coherence protocol is used. The user can also specify that certain data items are non-cacheable.

3.5 Write Buffer Module

A write buffer is used to buffer a write accesses, so that the processor requesting a write can proceed without waiting for the write to complete. Buffering can improve performance of the system when there is significant delay between processors and memory modules, due to network latency and/or contention. The write buffer sends the write transaction to the appropriate next module when the interconnection is free. It also keeps track of forwarded write transactions until an acknowledgment comes back. The write buffer has to be concerned with the ordering among reads and writes and when the accesses can be issued. The implementation of the write buffer largely determines the consistency model of the system. A brief description of

different consistency models are given in [20]. The weaker consistency models take advantage of the synchronization statements inserted by the programmer and attempt to keep the system consistent only at the synchronization points. The implementation of different consistency models in MEMSIM are confined to the write buffer module.

When MEMSIM is used without write buffer modules, the system is sequentially consistent. The write buffer module implements two types of consistency models, processor consistency and release consistency. When initializing the write buffer module, the user can choose the size of the buffer and the consistency model, and supply the module with a pointer to the function implementing the coherence protocol. The write buffer must refer to a coherence function since there can be a coherence request for a cache line that is waiting at the write buffer and the transaction type for this line could change as a result of that coherence request.

3.6 Bus Module

The bus module simulates the bus connecting multiple processor nodes. The bus module simulates the time to complete a transaction and the time spent waiting for the bus. It can also simulate a snooping coherence protocol when the user initializes this module with a pointer to a coherence function.

MEMSIM implements two type of bus modules, one that supports split transaction and one that does not. A non-split transaction bus sends a request to the next module (usually the memory) and the bus is held until the reply is sent back. A split transaction bus sends a request to the next module and does not wait for the reply. The bus may transfer another request while the memory module completes the access. The memory module will then have to arbitrate for the bus to send the reply back.

If the bus module is initialized with a pointer to a coherence function, the bus module has to also complete the coherence actions of each transaction (such as in-

validating or updating other caches). It is more inefficient to simulate snooping as it occurs in a real system since this involves each cache module having knowledge of every transaction on the bus. Instead, for each transaction the bus module updates a data structure that keeps track of all the caches that have a copy of each line. When the cache replaces a line, it calls a sub-routine to update the bus data structures. Therefore the bus module always has the current list of caches that have a copy of a line.

On each transaction the bus module calls the coherence routine of the bus with the transaction type. The coherence routine returns a list of caches (if any) that must receive a coherence message in order to complete the current transaction. For instance when implementing the Write-Back with Invalidate coherence protocol, a write request to a shared line will have to invalidate all copies of the line. When such a write transaction is sent on bus, the bus module sends an invalidation message to the caches on the list returned by the coherence routine. The write transaction completes only after all the necessary caches have been invalidated. This achieves the same result as a snooping coherence protocol.

3.7 Memory Module

The memory modules in MEMSIM simulate the time to access the module. The user can specify the time to complete a read or a write access to a word in the burst mode and in the individual access mode when the module is initialized. When the memory module is accessed, the data is always assumed to be present. That is, page faults are not simulated. If the user wanted to simulate virtual memory, some of the cache data structures, could be used to maintain page tables. Also the cache routines used to determine a hit or a miss in the cache can be adapted to determine the hit or miss of a page in memory.

3.8 Directory Module

The directory module is used in systems that implement directory-based coherence protocols. This module is used in systems where processors communicate with one another through an interconnection network that does not support an efficient broadcast mechanism. It may also be impossible to achieve atomic broadcasts in such systems. Therefore a directory entry is maintained for each cache line-sized block in memory. This entry maintains the state of the line and either a list of caches that have a copy of the line or a pointer to such information. A more detailed description of various directory schemes can be found in [6]

MEMSIM implements a full-map directory scheme. Each directory entry consists of a bit vector and some state bits. The size of the vector is equal to the number of processors in the system. Each bit in the vector is set or cleared depending on whether or not that particular processor cache has a copy of the line. Currently Write-Back with Invalidate coherence protocol is supported in this module. Other protocols can be supplied by the user as a pointer to a function when the module is initialized.

MEMSIM allows the user to map program data to a specific memory module. The user can also classify data as private, in which case it is assumed to be mapped to the local memory of that node. The user can map the program data by making calls in the user program to associate a certain address range with a particular memory module. From this information, the cache module determines the home node of a given request, when it misses in the cache. The request is routed to the appropriate directory module, and the directory module follows the coherence protocol to keep the data coherent.

3.9 Network Interface

NETSIM is a general-purpose interconnection network simulator. The Network Interface module is the interface between MEMSIM and NETSIM. This module essentially simulates the network ports. It converts MEMSIM packets into NETSIM packets and vice versa. When this module is created, the user must provide the buffer size of the network ports. There are two type of network interface modules in MEMSIM. One of them can be used to send packets into the network and the other can be used to receive packets from the network.

This module is specially designed to work with directory-based architectures that require two networks in order to avoid deadlocks. The necessity for two networks is described in the Section 4.5. One of the two networks is used to send memory or coherence requests to a remote node or to receive requests from a remote node to the memory or cache module in this node. The other network is used to send (receive) replies. Therefore each network interface module also requires pointers to two NETSIM ports when initialized.

3.10 NETSIM

NETSIM is a general-purpose interconnection simulator. It can be used to construct and simulate a wide range of network models, including both direct and indirect networks. It is designed to simulate large networks that use modern routing techniques, such as worm-hole and virtual-cut-through routing, but can also simulate networks that use store-and-forward routing. All the experiments described in this thesis were done for networks using worm-hole routing. The NETSIM Reference Manual [27] and a related paper [28] provide more details about this simulator.

Chapter 4

Experimental Setup

This chapter describes the architecture simulated. The architecture of the shared-memory multiprocessor studied is shown in Figure 4.1. The figure shows the mesh network, although we simulate different network topologies. A 64-node system is simulated, with each processing node consisting of a processor, a cache, a memory, a directory module, and a network interface. Each of these modules and the parameters pertaining to each module are described in the following sections.

4.1 Processor

As described in Section 3.3, the processor is not simulated in detail in an execution-driven simulation. The profiler for the SPARC architecture (which is not super-scalar) is used with all simulations presented in this thesis. The simulator keeps track of simulated time in terms of the number of processor cycles executed instead of in seconds. Therefore the processor cycle time is not directly relevant. However in order to determine the access time of other modules in the system in terms of processor cycles, it is necessary to determine the processor speed. We assume a processor speed of 50 MHz.

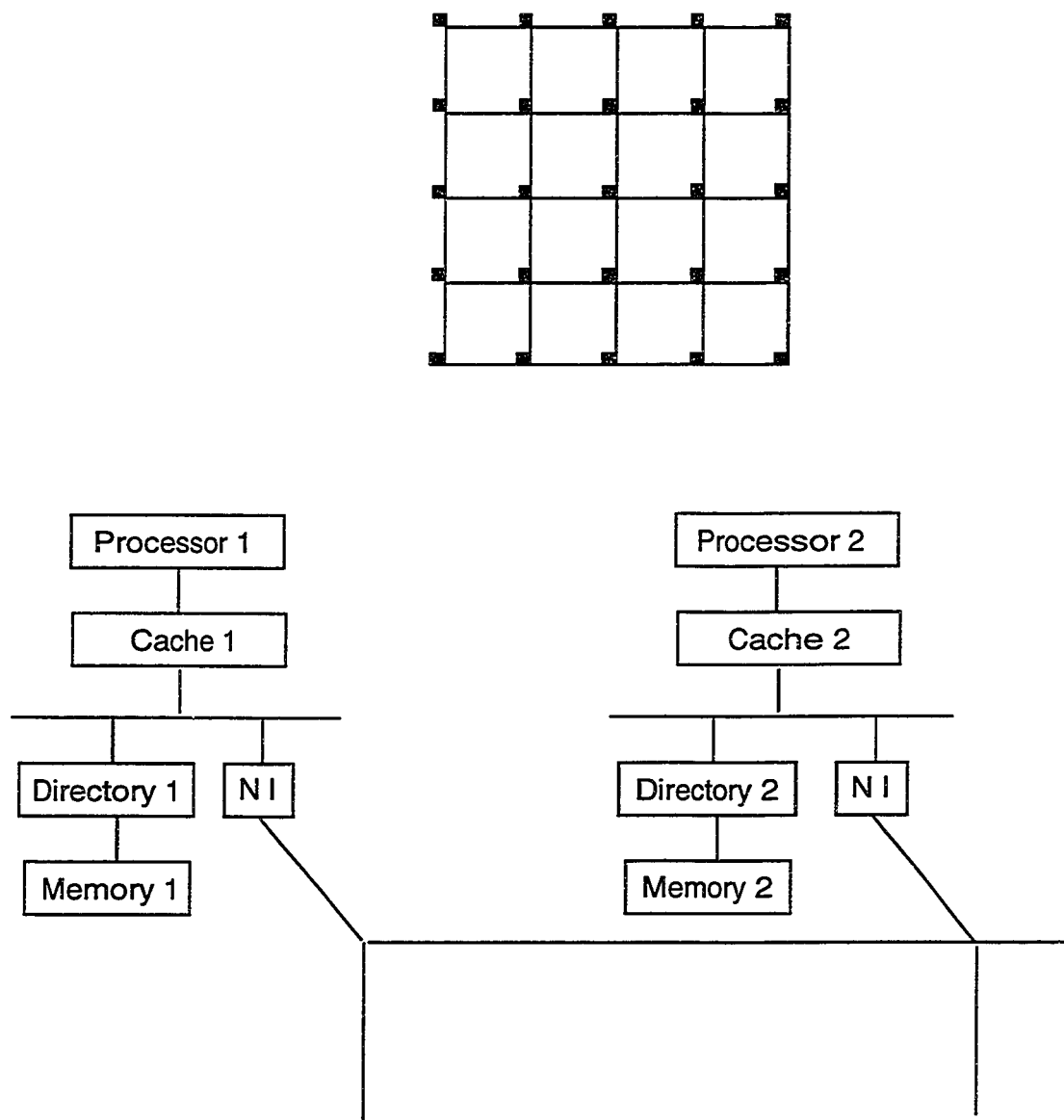


Figure 4.1: Scalable Shared-Memory Multiprocessor

4.2 Cache

Each processing node has one cache module. The caches are 32 KB in size, 2-way set associative and have a cache line size of 32 bytes. A survey of current workstations shows that cache sizes vary between 16 KB and 4MB, with 256 KB being the most common configuration. We chose to simulate a smaller cache, in order to scale down the problem sizes and be able to run experiments in a reasonable amount of time. Although larger set sizes in the cache can lead to better performance, it has been shown that the maximum performance gain is obtained when going from a direct-mapped cache to a set size of two [41]. Therefore we simulate a 2-way set associative cache. Only data accesses to the cache are simulated. We assume that all instruction accesses hit in the cache. This is a reasonable assumption, since experiments have shown that even small instruction caches have very high hit rates [42].

4.3 Memory

Each processing module has one memory module. When the memory module is accessed for a given address, the page is always assumed to be present. That is, virtual memory is not simulated. We simulate a memory module with a 60ns access cycle and a fast page mode access time of 40ns. This is equivalent to 3 processor cycles for the initial access and 2 cycles for each additional word accessed. The data paths inside each node are 64 bits wide. Therefore the memory module takes 9 cycles to return a 32-byte cache line on a read. For simplicity we assume that the writes take just as long. Wherever possible, the program data of applications is mapped to the memory module close to the processor that uses the data. In Section 5.1, when the applications used in this thesis are described, the data mapping used for that application is also described.

4.4 Directory

The directory module is used to maintain coherence in the system. A directory module is associated with the memory module in every node. Each cache line in memory has a directory entry associated with it. A full bit vector directory scheme is simulated in all our experiments. In this scheme, each directory entry consists of a vector and some state bits indicating the state of the line. The size of the vector is equal to the number of processors in the system. Each bit in the vector is set if the corresponding processor cache has a copy of the line. This type of directory scheme is not as scalable as chained directories, or a limited directory with software extensions. However, this scheme is the most efficient in terms of the number of messages sent and the overall access time. It has been shown that this scheme can be made more scalable by using a cache of directory entries instead of having a directory entry for every line in the physical memory [23].

Since the directory module is associated with the memory module, it will most probably be implemented in DRAM technology. Most accesses to the directory data structure are Read-Modify-Write (RMW). This can take as much as 115 ns in a DRAM with a 60 ns access time. This translates to approximately 6 processor clock cycles for each access. The directory also has to create and send out invalidation messages. We assume a packet creation time of 6 cycles for the first packet and 2 cycles for each additional invalidation packet that is sent out for the same request.

4.5 Interconnection Networks

The various parameters that relate to the interconnection network include the topology of the network, the bandwidth of the network, the size of buffers in the various switches within the network and in the ports of the network, the speed of the network and the type of routing.

The different network topologies that we studied are the mesh, the hypercube and the shuffle-exchange network. The performance of the multiprocessor is measured as the network topology is varied. The networks are compared for two cases: (i) all the networks have the same channel bandwidth, and (ii) all the networks have the same bisection width. The channel width is varied between 4 bits and 64 bits (8 bytes), for all three network topologies. The bisection width of a network is the minimum number of wires that must be cut to divide the network into two equal halves [13]. Bisection width is a measure of network cost since the complexity of a connection topology is wire-limited. A 64-node mesh (8 x 8 grid), has a bisection width of 16 channels. A 64-node hypercube (6 dimensions), has a bisection width of 64 channels. A 64-node shuffle-exchange network, with a switch degree of 4 and a depth of 2, has a bisection width of 32 channels. In order to have the same bisection width in all network topologies, the channel width of the mesh can be four times the channel width of the hypercube and twice the channel width of the shuffle-exchange network.

We use the wormhole routing technique to route packets between nodes. For the mesh and the hypercube network we use the e-cube routing algorithm which is guaranteed to be deadlock-free and is deterministic [35]. The routing in the shuffle-exchange network is also deterministic. Deadlock-free adaptive routing algorithms have been proposed for wormhole routing [35]. These algorithms use extra channels. These may be extra physical channels or multiple virtual channels that share one physical channel. Both of these options are more expensive to implement when compared to deterministic routing. It remains to be seen whether adaptive routing gives better performance for realistic workloads.

The e-cube routing is dimension-ordered routing. For example, in the mesh network a packet is first routed in the X-direction until it reaches some node on the column on which the destination node is located. Then the packet travels in the Y-direction until it reaches the destination. In the case of the hypercube there are two

nodes in each dimension. Starting with the lowest dimension, the packet is routed to the other node if the bits in the position corresponding to the current dimension in the source and destination node address differ. In the shuffle-exchange network the routing algorithm depends on the switch size. With a switch size of 4, the routing algorithm examines the destination address 2 bits at a time starting with the most significant bits first. The packet is routed to the output link selected by the two bits that is examined.

The network ports can buffer four packets. The directory module has buffer space that is used for packets waiting for a service to be completed, and also for packets waiting to enter the network when the port buffers are full. This in effect increases the port buffer space. In all our experiments we use a buffer space of 64 packets in the directory modules. The switch buffer size used in all experiments is 2 flits.

The speed of the network is assumed to be half that of the processor. That is, it takes two processor cycles to transfer a flit between two switches. The architecture simulated uses a pair of interconnection networks. One of them sends request packets and other receives replies. The two networks are necessary in order to avoid deadlocks. The network and the routing protocol in the network are guaranteed to be deadlock-free if the messages are consumed at the output. However, due to limited buffering in the directory module it cannot be guaranteed that the messages will always be consumed. The use of two networks avoids deadlocks by the following mechanism. First, the reply messages are always consumed because they are allocated dedicated buffer space when the request is sent out. Second, the request messages are serviced if there is buffer space in the module. The request messages are sent back as a reply with a negative acknowledgment if there is no buffer space in the module. Since there is not an infinite supply of requests (requests need to pre-allocate buffer space to receive the replies), and requests are turned into replies if they cannot be serviced

due to lack of buffer space and replies are always received from the network, there cannot be a deadlock.

The use of two physical networks can be avoided by multiplexing packets on a single network or by dropping packets that cannot be delivered and using a time-out mechanism to re-send these packets. Neither of these options are used in our simulation for the sake of simplicity.

4.6 Coherence Protocol

The coherence protocol simulated is write-back with invalidate. All simulations are sequentially consistent. It has been shown that better performance can be obtained with weaker consistency models. We feel that the performance of the network under sequential consistency should be studied, in order to understand the performance of the network when using various weaker consistency models. The study of the effect of network performance for a weakly consistent system is for future work.

The Figure 4.2 shows the states of a line in the cache and the transactions involved. Each line in the cache can be in one of five states: Invalid (Inv), Shared-Clean (ShCl), Private-Dirty (PvtDy), Shared-Clean Pending (Pnd_ShCl), and Private-Dirty- Pending (Pnd_PvtDy). When the application is started all cache lines are invalid. The Figure 4.3 shows the states of a line in the directory module and the transactions involved. Each line in a directory module can be in one of four states: Uncached, Shared, Dirty, and Pending.

In both figures the transactions are labeled as follows: a prefix P_ is used when the transaction originates at the processor. The prefix N_ is used when the transaction originates at the cache and is sent through the network to the directory and the corresponding acknowledgment comes back from the directory. The prefix C_ is used for all coherence requests originating from the directory module and sent to the caches.

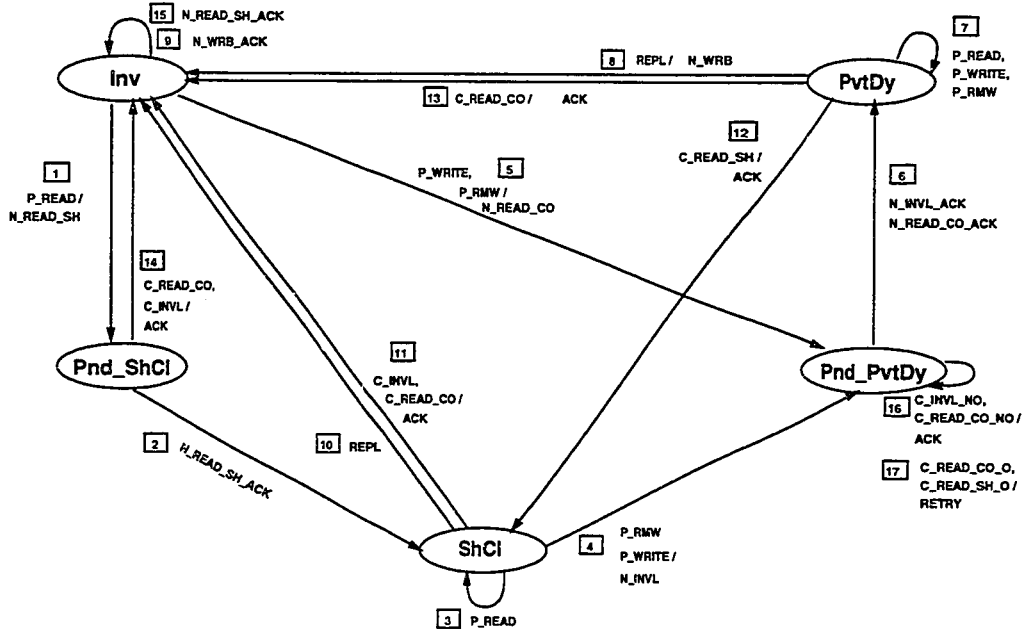


Figure 4.2: State Diagram of the Write-Back Coherence Protocol

All transactions are labeled either X/Y or X , where X is the incoming request, and Y is the response.

The cache lines are initially invalid. On a read from the processor, a read request is sent to the directory module. When the read transaction is completed, the line is in the state Shared-Clean. On a write or a read-modify-write access from the processor, a write transaction is sent to the directory module. If a line in the directory module is in Uncached state, the read or the write transaction completes immediately. If the line is in Shared state, a read transaction can complete, whereas a write transaction will wait until all the copies of the cache line are invalidated. If the line in the directory module is in Dirty state, then the read or the write access cannot complete until a coherence access to the current owner of the line is completed. This access fetches the most recent copy of the data and also updates the state of the line at this cache to Shared-Clean or Invalid in the case of read or a write access respectively.

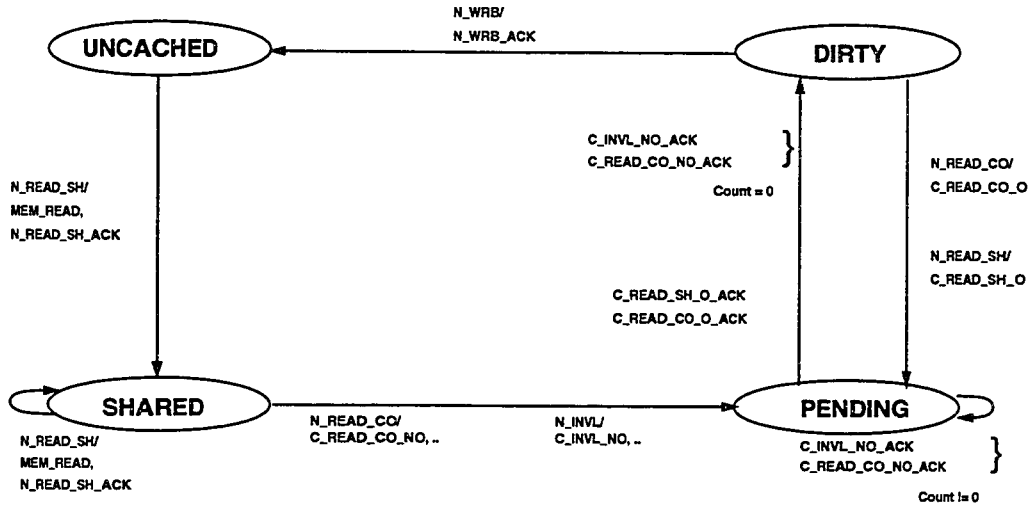


Figure 4.3: State Diagram Showing Different States of a Directory Line

There are certain coherence accesses that can happen in a system with a directory-based coherence protocol that will not happen in a snoopy coherence protocol implemented on a bus. This is due to the lack of global knowledge in a system with distributed memory. When a cache line load is pending from a read access, the cache can get a coherence access to invalidate the line. This can happen if the network does not guarantee ordered delivery, which is not the case in our simulation. This can also happen due to the presence of two networks in the systems. In this case the line is invalidated. When the reply arrives, the pending read is allowed to proceed but the line is not loaded in the cache. Any subsequent access to the same line would miss because the line was already invalidated.

When the cache is pending due to a write access, it can get an invalidate request. The directory might have sent the invalidate request before the directory processed this pending write or after. In the coherence protocol simulated, the directory tags the request such that the cache can determine whether this request came before or after the directory sends a reply to this module. In the former case it acknowledges

the coherence request and does not change its state (transaction 16 in Figure 4.2). This is because, the cache is certain that when the pending write returns the cache state will be restored. In the case that the request to invalidate the cache line was sent by the directory after the reply to this cache was sent out, the cache cannot acknowledge the invalidate access (transaction 17 in Figure 4.2). This is because the pending write access will change the data in the cache line. In this case the cache sends a message to the directory to retry the invalidate access.

This chapter has described the various parameters used in the simulations. The architecture simulated has 64 nodes, with a processor, a cache, a directory, and a memory module in each node. The system supports a write-back with invalidate coherence protocol, a full bit map directory, and sequential consistency.

Chapter 5

Results

This chapter presents the results of our study on the effect of the interconnection network on the performance of shared-memory multiprocessors. The architecture simulated and all the parameters of the architecture were presented in Chapter 4. Four network topologies: the mesh, the hypercube and two shuffle-exchange configurations, are studied at various network bandwidths. Section 5.1 gives a brief description of the applications used in our study. The following sections present the results obtained and a discussion of these results.

5.1 Benchmarks

The benchmarks used should be representative of the projected workloads for the described architecture. The architecture is a scalable shared-memory multiprocessor. We expect the architecture, to be used like current message-passing multiprocessors, for compute intensive applications. The following algorithms were chosen from the numerical and non-numerical domains of study: Matrix Multiplication, Successive Over Relaxation, Fast Fourier Transform and Sorting.

Most of these algorithms use barriers for synchronization. Tournament barriers were implemented since they have relatively lower latency for the completion of the barrier ([24]). Conceptually, to achieve a barrier using this algorithm, processors start at the leaves of a binary tree with a fan-in of two. One processor from each node continues up the tree to the next “round” of the tournament. The processor reaching the root of the tree writes to a global flag on which all the other processors

are spinning. By contrast, the central barrier is implemented by having all processes increment a single global counter, which causes a lot of network traffic.

5.1.1 Matrix Multiplication (MMULT)

The multiplication of two matrices involves mostly read-only sharing of data. Each process calculates the result for a square portion of the result matrix. Each process reads an eighth of each of the two input matrices in order to calculate a sixty-fourth of the result matrix. In this program one of the input matrices is transposed before the multiplication in order to increase locality in the caches. This way both matrices are accessed across rows. Multiplication of two 128x128 matrices is simulated.

The portion of the result matrix that is computed by each processor is also allocated to the memory module associated with that node. The two input matrices are allocated in similar sized blocks to each node. About seven-eighths of the total portion of the two input matrices read by a process are allocated on nodes that are remote to that process. A single barrier is used in the program to synchronize the processes after the transpose is completed.

5.1.2 Successive Over Relaxation (SOR)

This is an iterative method of solving partial differential equations (see [38] for a description of the algorithm). Each computation depends only on its nearest neighbors. Hence active sharing is limited to the boundary elements. The program partitions data by blocks of rows. This program uses two matrices. During each iteration the data is read from one matrix and the results written to the next. During the next iteration the roles of the two matrices are switched, reading from the matrix previously written to and writing to the other matrix. We ran this program for a 256 x 256 matrix. Each data item is a double-precision floating point value. We ran

ten iterations of the program. While this is not enough to obtain convergence, the behavior of the algorithm is identical in all iterations after the cache has been filled.

The two matrices are partitioned by rows and allocated to the different memory modules. Four rows of each matrix are allocated to each memory module. Data is shared between neighboring processors in SOR. The synchronization is achieved by the use of locks. There is a lock associated with each shared row. Each lock is accessed by only two processors. This makes the lock-based implementation faster than having a barrier at the end of each iteration.

5.1.3 Fast Fourier Transform (FFT)

The FFT program is based on the Cooley Tukey Radix-2 Decimation in Time algorithm. A general description of the algorithm can be found in [38]. The computation has $\log_2 N$ stages, where N is the number of data points for which the FFT is computed. The processors synchronize with the help of a barrier at the end of each stage, for the first $\log_2 P$ stages, where P is the number of processes doing the computation. The last $\log_2 N - \log_2 P$ stages do not need synchronization since each process operates on the same data in every stage thereafter. Each data point consists of two double precision numbers, one for the real part and the other for the imaginary part. The experiments in this thesis were conducted for a data set size of 32,768 (2^{15}) complex numbers.

The program uses two data sets, one of them holding the actual numbers to be transformed and the other holding a pre-computed set of sine values that is used during the computation. The entire data set is divided into sixty-four portions and the first portion is allocated to node zero and so on. During the first $\log_2 P$ stages of the program, the processes are computing the transform on data that is not allocated at the local node. In this phase there is a lot of data movement across the network. During the remaining $\log_2 N - \log_2 P$ stages of the FFT, the processes are operating

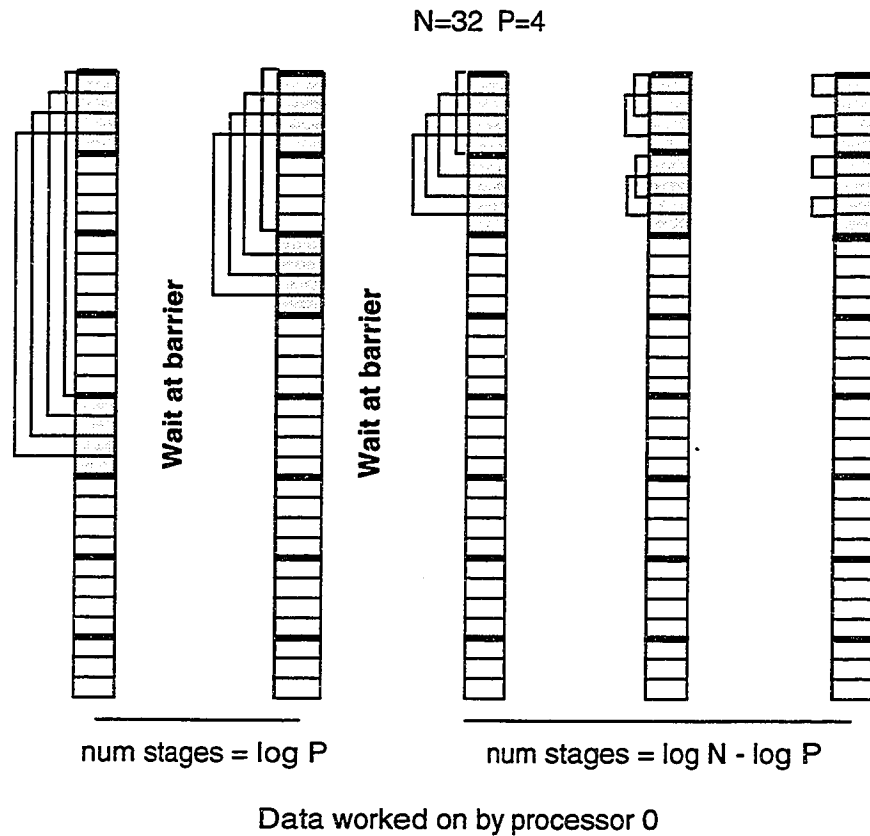


Figure 5.1: Communication Pattern of the FFT Algorithm

on data allocated to the local node. If this data fits in the cache, it will have a very high hit rate. Even if it does not, the miss penalty will be quite low. Figure 5.1 shows the communication pattern of the algorithm. One can see how this would map very well on a hypercube. The figure does not show the communication pattern of the pre-computed sine values.

5.1.4 Sorting (Bimerge)

Each process in this program sorts a pre-allocated sublist. The processes then proceed to merge the sorted sublists two at a time. A description of the algorithm can be found in [43]. During the merge phase all processors participate in the merging instead of a naive algorithm where in each subsequent merge phase only half the number of processors are involved in merging two previously merged sublists. The merge phase involves $\log_2 P$ stages and there is a barrier synchronization after each stage. During each merge phase, the two lists that are being merged are split into as many sublists as there are processors participating in that merge phase. Then the cross over points between the sublists are determined and a parallel algorithm determines a pair of sublists that need to be merged. The merging of the pair of sublists that crossover is straightforward. We ran simulations for a data set size of 65,536 (2^{16}) data points.

The data set is partitioned into 64 portions. The first portion is allocated to the memory module in node 0, etc. Each processor first sorts the array local to that processor. During the merging the amount of data communicated and the distance traveled by the data is dependent on the data itself. Since the merge phase does not update in place, there are actually two arrays used in the program.

The Table 5.1 gives a brief description of each program and the data set size.

Application	Problem Size	Shared Data Space	Program Description
MMULT	128 x 128	196608 bytes	Matrix Multiplication
SOR	256 x 256	1048576 bytes	Successive Over Relaxation
BIMERGE	2^{16} (65536)	524288 bytes	Merge Sort
FFT	2^{15} (32768)	786432 bytes	Fast Fourier Transform

Table 5.1: Applications Used in Study

5.2 Effect of Network Topology

This section presents the results of our study on the effect of the interconnection network topology on the performance of shared-memory multiprocessors. A 64-processor system was simulated. The network topologies studied were the hypercube, the mesh and the shuffle-exchange network. We ran experiments for two types of shuffle-exchange networks. In one case each switch was a 4x4 crossbar (SE4). This reduced the total number of stages in the network to 3. The other network used a 2x2 crossbar switch (SE2), making the total number of stages equal to 6 for a 64-processor network. These particular topologies were chosen because we feel that they represent three very different types of networks. The mesh and hypercube are direct networks. They represent two ends of the spectrum of the group of networks called direct k-ary n-cubes. The shuffle-exchange network is an indirect network. It was chosen to see how it compared to some direct networks. The switch size of the network in a shuffle-exchange network trades contention against latency. We simulated two different switch sizes to see how this fares in a shared-memory multiprocessor. The performance of multiprocessors having different network topologies, each with constant channel width, were studied. The performance of the multiprocessor when the networks have a constant bisection width was also investigated.

5.2.1 Simulation of Networks with Constant Channel Width

The Figures 5.2 to 5.5 show the performance of the four different programs for different network topologies. Each graph has four curves showing the performance for the hypercube, mesh and the two shuffle-exchange networks. Each curve shows the execution cycles taken for each network topology when the channel width is varied between 4 bits and 64 bits. Table 5.2 shows the relative performance of these applications at a channel width of 4, 16 and 64 bits for the different network topologies.

	Program	HCube / Mesh(%)	HCube / SE4(%)	HCube / SE2(%)	SE4 / Mesh(%)	SE2 / Mesh(%)	SE4 / SE2(%)
Link Width 4 bits	FFT	56.90	15.23	12.82	26.56	28.09	-2.09
	BIMERGE	14.00	18.03	11.44	-3.53	2.25	-5.58
	SOR	0.85	24.32	2.93	-23.28	-2.07	-17.20
	MMULT	2.28	2.33	4.09	-0.04	-1.73	1.68
Link Width 16 bits	FFT	41.72	4.71	9.48	26.11	22.75	4.50
	BIMERGE	3.60	2.56	4.87	1.00	-1.23	2.20
	SOR	1.57	0.75	2.48	0.81	-0.90	1.72
	MMULT	0.57	0.05	0.12	0.51	0.44	0.07
Link Width 64 bits	FFT	18.47	-2.91	6.38	18.04	10.20	9.56
	BIMERGE	0.26	-1.06	4.34	1.32	-4.70	5.46
	SOR	2.31	-1.94	0.99	14.15	1.29	2.99
	MMULT	0.70	0.01	0.01	0.68	0.68	0.00

Table 5.2: Comparing the Performance of Shared-Memory Multiprocessors with Different Network Topologies

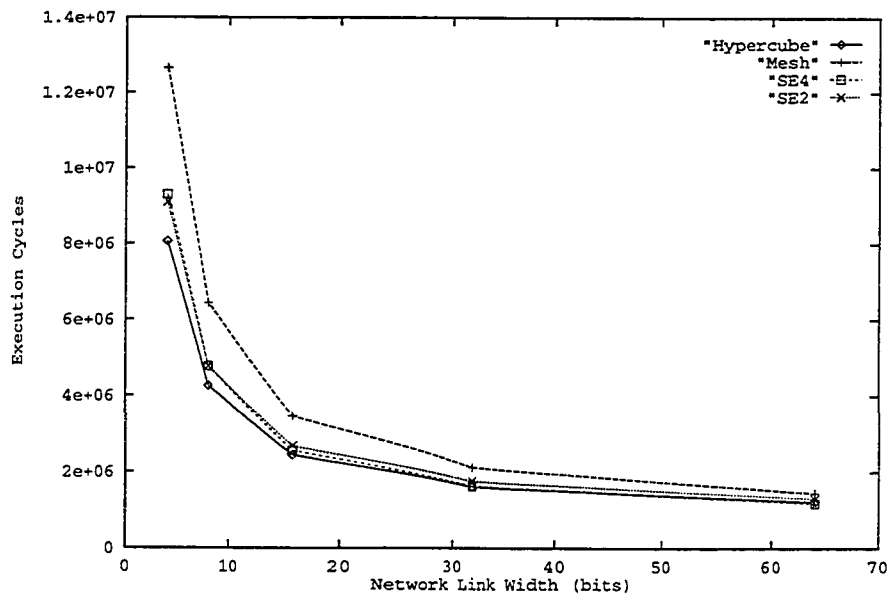


Figure 5.2: Performance of Different Network Topologies for FFT

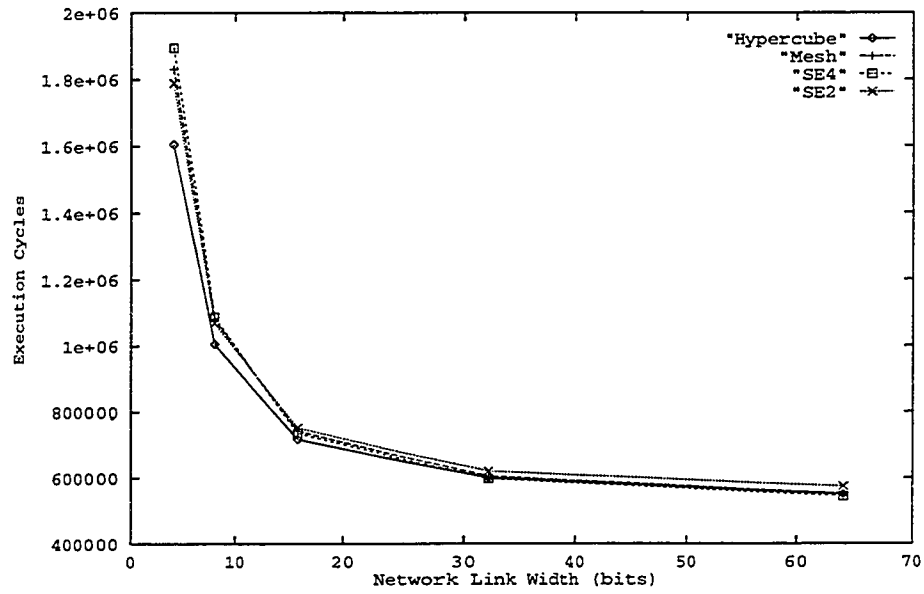


Figure 5.3: Performance of Different Network Topologies for Bimerge

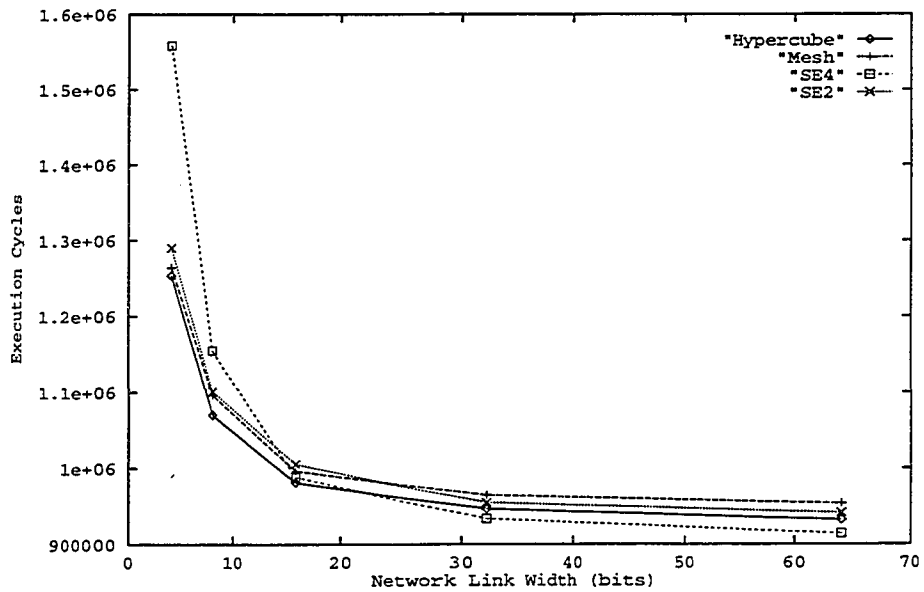


Figure 5.4: Performance of Different Network Topologies for SOR

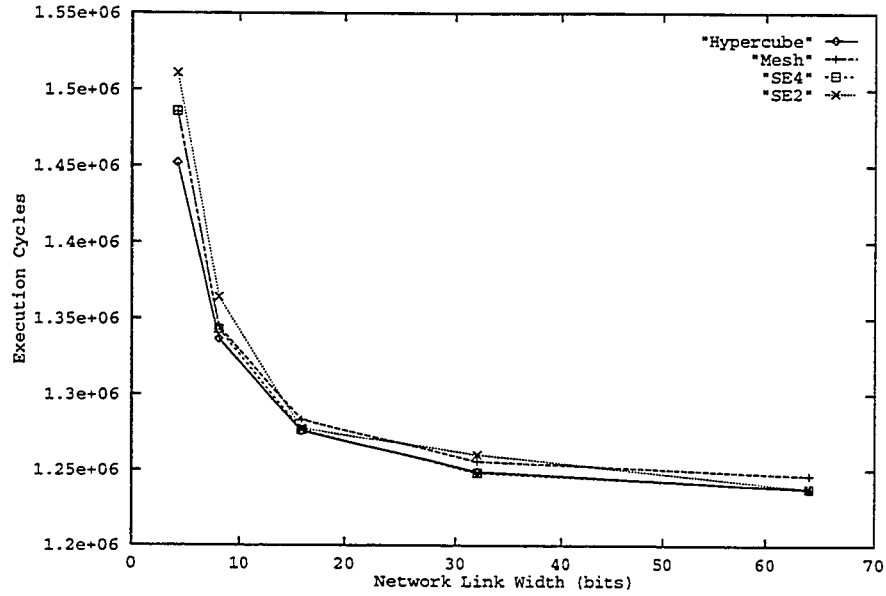


Figure 5.5: Performance of Different Network Topologies for MMULT

The following paragraphs first give an overview of the trends seen in the graphs. Then we provide a detailed explanation on a per application basis as to the causes of these trends.

The following is obvious from examining the four graphs

- Overall the hypercube network does better than the other two networks. Even when SE4 does better than the hypercube, the difference in performance is under 3%.
- The differences among the three networks are the most pronounced at a link width of 4 bits.
- When the link width goes up to 16 bits and higher, in three out of the four applications the differences in performance among the networks reduce to less than 5%.

Hypercube Versus Mesh: A hypercube network with the same channel width as a mesh network has a comparatively lower diameter and higher bisection width. However, only two out of the four programs that we ran, FFT and Bimerge, showed significant improvement when using a hypercube network over a mesh. SOR and MMULT show almost no performance improvement in using the hypercube. This is because the FFT and Bimerge have communication patterns that are able to use the hypercube effectively. SOR, on the other hand has nearest neighbor type communication. In the case of MMULT, every eight processors read the same sixteen rows of the two input matrices, which is distributed evenly across the eight memory modules. The communication pattern of MMULT for the hypercube and the mesh networks are shown in Figures 5.8 and 5.9 respectively. This pattern does not map well on any of the network topologies. The amount of communication with respect to computation is also lower for MMULT. Therefore, MMULT performance is almost the same in all the networks.

Hypercube Versus Shuffle-Exchange Network: The SE4 does worse than the hypercube when the channel widths are small. The difference in performance is significant for the FFT, Bimerge and SOR. The hypercube network is better able to handle contention. When channel width is increased to 32 or 64 bits, the shuffle-exchange network does slightly better than the hypercube. This is because at this bandwidth there is hardly any contention in the network. The number of hops taken by the packet is higher on the hypercube for some applications and due to the difference in latency the shuffle-exchange network starts performing better. This shows that the SE4 performance degrades faster than that of the hypercube as the bandwidth is reduced.

The SE2 also performs worse than the hypercube for all of the applications. The difference in performance is significant for FFT and Bimerge. Unlike the SE4 network, the SE2 network never outperforms the hypercube. The maximum number of hops

in the hypercube is 6. The average number of hops can be expected to be lower. The number of hops taken by a packet in this shuffle-exchange network is 5. With negligent contention in networks with higher bandwidths, the network latency depends on the size of the packet (same for all network topologies) and the number of hops taken by the packet.

Shuffle-Exchange Network Versus Mesh: There is a significant difference in performance between these two networks only for FFT and SOR. In the case of the FFT, SE4 and SE2 both perform better than the Mesh. In the case of SOR the mesh network performs better. We will discuss this program dependent behavior in the following paragraphs.

FFT: As seen in Figure 5.2, the hypercube out performs all the other networks. The mesh network performs the worst. Analyzing the program behavior, we find that a significant fraction of the misses are due to accesses to the sine and cosine values. A single array is initialized at the start of the program and the sine values are accessed from the beginning of this array and the cosine values start at a quarter of the way down this array. During the first stage a process reads consecutive locations of the sine and cosine array. Therefore each cache line fetch services four sine or cosine values. During the next stage the process reads every other sine value and so on. In the final stage it is reading values 16 K apart in the array and there are only 32 K data items. At this point there is high reuse of data. However, in the middle few iterations the process has to access almost all the nodes in order to complete all the sine and cosine accesses. Out of the four applications, FFT has the lowest cache hit rate (84%).

The request of each process to fetch these sine and cosine values is distributed across nodes 0 through 47. Since the latency of a cache line fetch in the case of a miss varies between 1600 cycles (channel width of 4) and 60 cycles (channel width of

64), it might be more efficient in this architecture to calculate the sine values instead of using pre-computed values.

There is also remote accesses for the actual data on which the transforms are calculated. In this program, there is a large amount of data movement in the first $\log_2 P$ stages of the program. In each stage each process may work on data that is completely different from the data it worked on in the previous stage. The data is also dirty at some other cache node and has to be fetched from there before the process can proceed. There is a barrier at the end of each stage. After a barrier most processes have a miss and they all send out a request at the same time, loading the network. Due to the global nature of the data accesses (i.e. each process sends data requests to almost every memory module in the system), the mesh network performs poorly for this application. The shuffle-exchange does not perform as well as the hypercube, because the accesses are not equally distributed to all nodes. The cosine array starts a quarter of the way into the entire array. Sine values are accessed from nodes 0-31 and cosine values are accessed from nodes 16-47. Therefore there are more accesses to nodes 16-31. Even among these nodes access is not uniform. The shuffle-exchange network suffers from hot-spot contention.

Bimerge: As seen in the Figure 5.3 the hypercube network performs better than the other networks. The SE4 does slightly worse than the mesh at lower channel widths. The SE2 does slightly better than the mesh at the lower channel widths. Both of these trends reverse at higher channel widths. But mesh, SE4 and SE2 network performance is within 5% of each other.

The amount and distance of data movement in this application is dependent on the data. For all the simulations the same data set was used, by using the same random number generator with the same seed. Due to the global communication pattern in this program, we expected the shuffle-exchange networks to perform better than the mesh networks as it did in FFT. But this was not the case. Examining the simulation

more closely we find that memory module 0 has four times as many accesses as any other module. The program uses some index arrays that are the size of the number of processors. We did not allocate these small data structures to a specific node and by default it was allocated to node 0. We could not allocate one element of each array to a specific memory module, since the data has to be allocated on cache line size blocks in the simulation. In a real system we expect this to be allocated in page size blocks, in which case this particular program would run as it did in our simulation. Since all nodes are trying to reach node 0, the shuffle-exchange networks experience hot-spot contention and the performance is not as good as the mesh. Pfister and Norton show in [36] that hot-spot traffic in multistage networks can produce effects that severely degrade all network traffic. This effect is shown to be independent of switching mode and topology of the multistage network.

SOR: As seen in the Figure 5.4, the performance of the hypercube, mesh and SE2 networks are close. The Figure 5.6 shows the nearest neighbor communication on 64-processor mesh. While most network accesses take one hop, accesses from processes on the edge of the mesh have to take 8 hops to get to its nearest neighbor. The Figure 5.7 shows the communication pattern of the SOR on a hypercube network. The figure shows a 8-processor sub-cube.

The performance of SE4 is worse than mesh, hypercube and SE2 for lower channel widths. As the channel width increases, SE4 starts to perform better than mesh and SE2. This application has nearest neighbor communication and uses locks for synchronization. The locks are also shared only by nearest neighbors. This application maps very well on the mesh network. Therefore in this case the mesh network performs as well as the hypercube network. In fact the average number of hops taken by a packet is slightly higher for the hypercube than the mesh. However, this nearest neighbor communication affects the SE4 very adversely. There is a high level of contention in each switch for the smaller networks (4 bits and 8 bits). This network

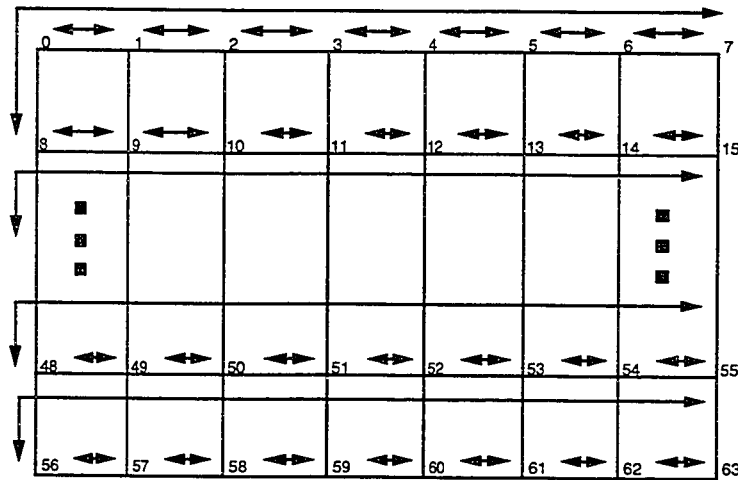


Figure 5.6: Communication Pattern of SOR on a Mesh Network

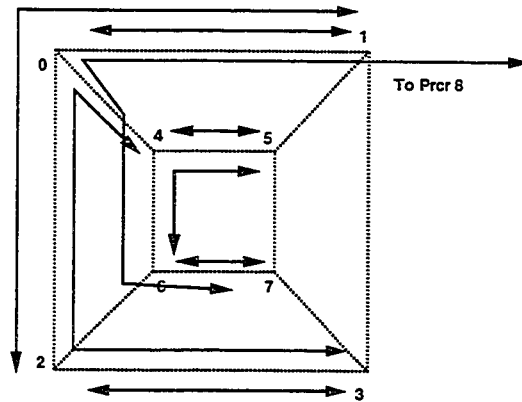


Figure 5.7: Communication Pattern of SOR on a Hypercube Network

does worse than the mesh network for these two cases. In SE4, all messages going to the first quarter of the destination nodes (nodes 0-15), leave the first stage switch on link 0, the second quarter (16-31) use link 1 and so on. With nearest neighbor communication, all four input processors at a switch will want to take the same link out, causing contention. In SE2, only two processors contend at each switch.

It is interesting that the shuffle-exchange network starts performing better than the mesh at a channel width of 16 bits. The average number of hops in the mesh is 1.9, which is slightly less than that of the shuffle-exchange. Our simulation shows that the average network time for the mesh is less than that of SE4 while the standard deviation is higher. Also, the average number of accesses is lower in the simulation with the SE4 network, implying that the processes spin-waiting on a lock gain access to the lock faster in the case of the simulation with the SE4 network. Therefore, it starts performing better than the mesh network.

MMULT: As seen in Figure 5.5, SE2 performs worse than the other networks. SE4 and MESH are clustered close together, and as usual the hypercube network does better. However, the difference in performance between hypercube and SE2 is only 4% even at the 4-bit link width. Figure 5.8 shows the traffic pattern to fetch one row of the first input matrix in a 8x8 mesh network. The figure shows only the first row of the mesh. Each arrow in the figure involves a fetch of 16 data values or two 32-byte cache lines. Figure 5.9 shows the traffic pattern in a hypercube network. The figure shows the first 8 nodes of the 64-node hypercube and shows only the fetch from processors 0 and 1. It should be noted that all the communication is within a sub-cube of size 8. The connectivity of the hypercube is not useful in this case. Due to deterministic routing the Z-dimension of the sub-cube is congested.

The traffic pattern for this program does not map particularly well on any one network. SE2 does worse than the other networks because of the higher number of hops. At a link width of 8 the differences among the different networks are negligible. The reason these differences seem more inflated in the graph is because MMULT has the least gain in increasing the link width. Therefore the range of the y-axis in the MMULT graph is lower than in the other graphs. We will discuss the performance of different link widths in Section 5.3.

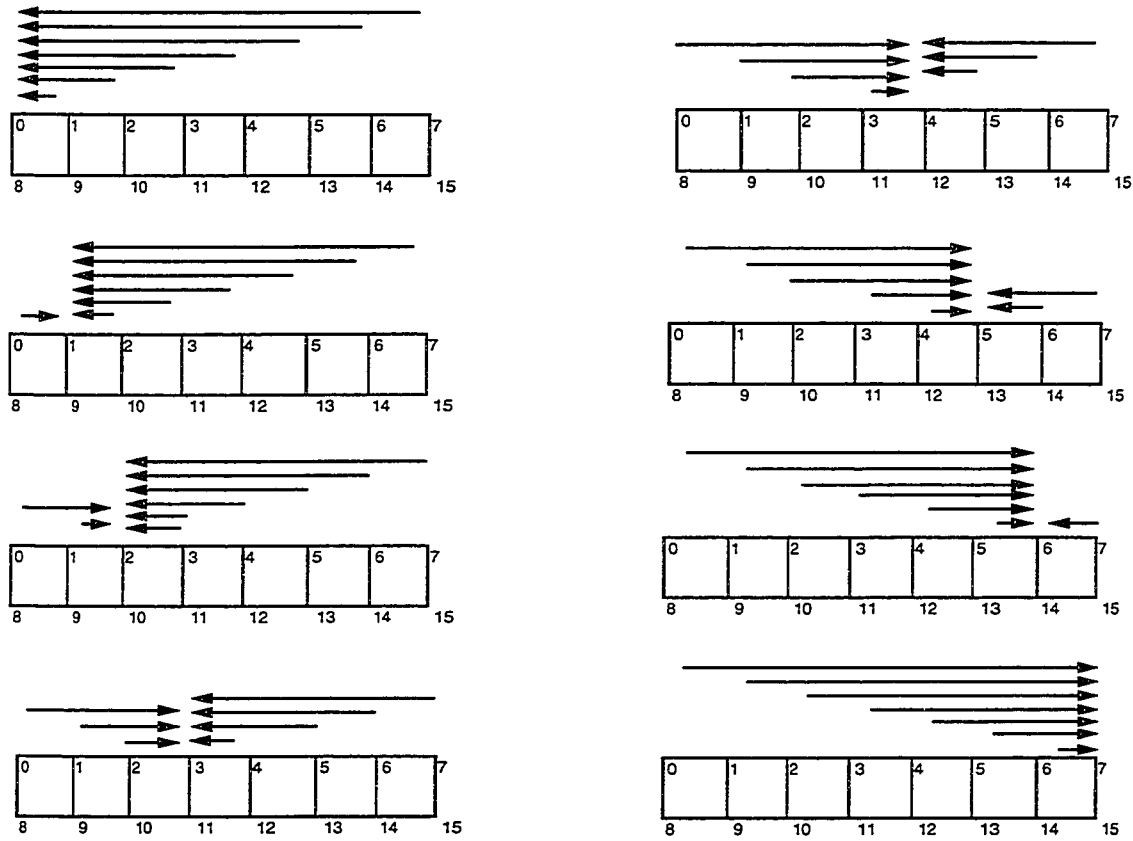
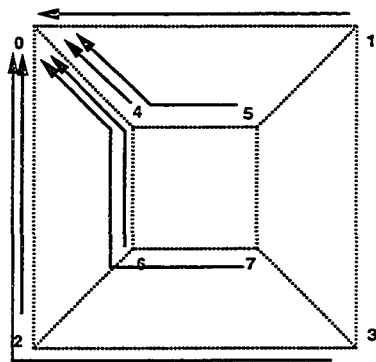
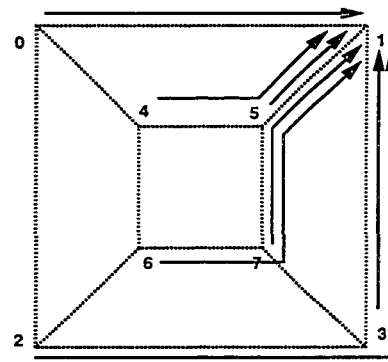


Figure 5.8: Communication Pattern of MMULT on a Mesh Network



Fetch From Processor 0



Fetch From Processor 1

Figure 5.9: Communication Pattern of MMULT on a Hypercube Network

In all our programs at least a sixty-fourth of the entire data set fit in the cache. If the program accesses the entire data set then it is going to have some misses. Also, care was taken to place the data close to where it might be used. This was done wherever obvious and for the larger data sets. Whenever a significant performance penalty was observed in the program, an effort was made to correct this. Examples include, the use of locks instead of barriers in the SOR program, the transposition of one of the input matrices in MMULT, and the use of an FFT program that was analyzed and optimized for shared-memory multiprocessors.

We did not implement an adaptive caching scheme which could potentially improve performance. In such a scheme, each data object in the program is associated with a different coherence protocol that is most efficient for the sharing pattern of that object. We did not implement this because in general it would take a significant effort by the programmer to analyze the program and associate certain data objects with a particular coherence protocol. Also, few systems currently support adaptive caching protocols.

Even using such optimized programs the hypercube network performs better. We believe that in the use of a shared-memory multiprocessor, there will be some programs that do not always hit in the cache and programs that are not optimized to perform well for that particular architecture (in the interest of having portable programs). In such cases we expect the difference in performance to increase.

In order to test this we looked at execution times for the matrix multiply algorithm in which the second input matrix was not transposed. In this case the hit rate of the cache is very low (66%). This is because the second input matrix is accessed column-wise. By contrast the hit rate when the matrix is transposed is 99.4%. Figure 5.10 shows the performance of the application for three network types: hypercube, mesh and SE4. The performance difference between hypercube and mesh is 59% at a link width of 4. The hypercube also performs better than the SE4 by 17 %. For

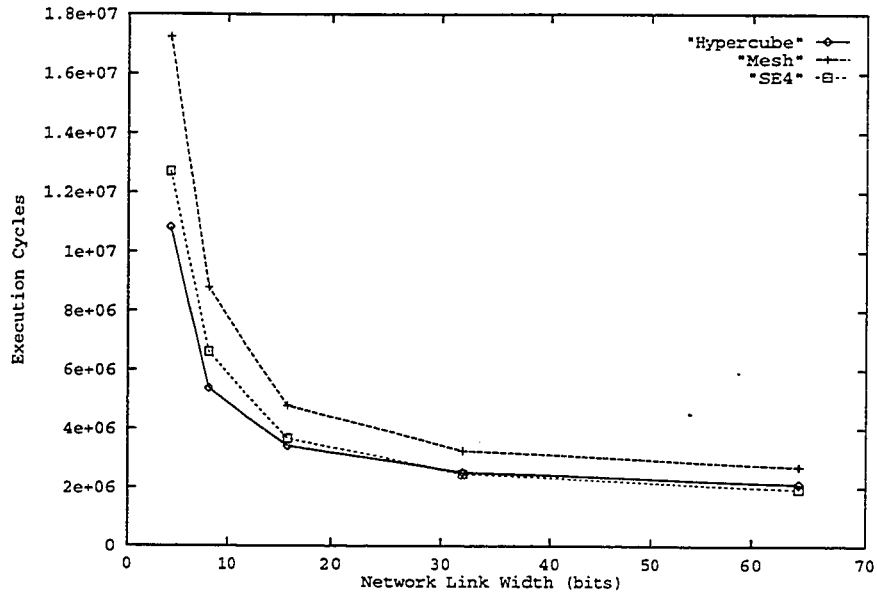


Figure 5.10: Performance of Different Network Topologies for Inefficient MMULT

the optimized program the improvement in performance with the hypercube was 2% compared to both mesh and SE4. The hypercube network is better able to tolerate the sustained high load. The performance are quite disparate even at a 64-bit link width (28% performance improvement for the hypercube over mesh).

We summarize the results for the effects of different network topologies when the channel bandwidth is constant as follows:

- The performance difference between the different networks is primarily due to difference in throughput or contention in the network. The average number of hops taken by a packet is 5 or below for all applications in all networks. The difference, between various network topologies, in average number of hops is 3 or less. We expect this difference to increase as the number of processors in the system increases, also increasing the performance difference.

- The hypercube network outperforms all the other networks.
- The difference in performance is significant only for 4-bit and 8-bit channel widths for three of the programs.
- The performance of a shuffle-exchange network with a switch size of 4 is susceptible to hot-spot contention when there is nearest neighbor communication or when all processors are trying to access a single memory module.
- The shuffle-exchange with a switch size of 2 is not as susceptible to hot-spot contention as the one with a switch size of 4.
- We expect these performance differences to increase when the cache hit rate is lower, due to capacity misses or because the application is not optimized for this architecture.
- The latency of a cache miss varies between 1500 cycles and 50 cycles for the different programs at the various channel widths. Due to these large latencies, the network performance affects the system performance significantly even when the cache miss rate is quite low.

5.2.2 Simulation of Networks with Constant Bisection Width

In the previous sub-section we compared the performance of shared-memory multiprocessors with mesh, hypercube and shuffle-exchange networks. The cost of building each of these networks is very different. The bisection width is a measure of the network cost. Since VLSI is wire-limited, the bisection width accounts for the wire density of the network. In this case assuming the width of the channel is 1 bit, the bisection width of the mesh network is 16, the bisection width of the hypercube is 64, and the bisection width of SE4 and SE2 is 32. In order to make bisection widths equal, the width of each link of the mesh must be four times the link width of the

hypercube and two times the link width of the SE2 and SE4. Table 5.3 shows the width of the mesh and the link width in other networks.

Figures 5.11 to 5.14 show the comparison of network performance of the different networks when the bisections widths are constant. The x-axis in the figure shows the link width of the mesh network. As seen in these figures when the cost of the network is considered, the mesh network performs much better than the hypercube or the shuffle-exchange network. The difference in performance is least when the mesh

Mesh	HCube	SE4	SE2
8	2	4	2
16	4	8	4
32	8	16	16
64	16	32	32

Table 5.3: Equivalent Link Widths

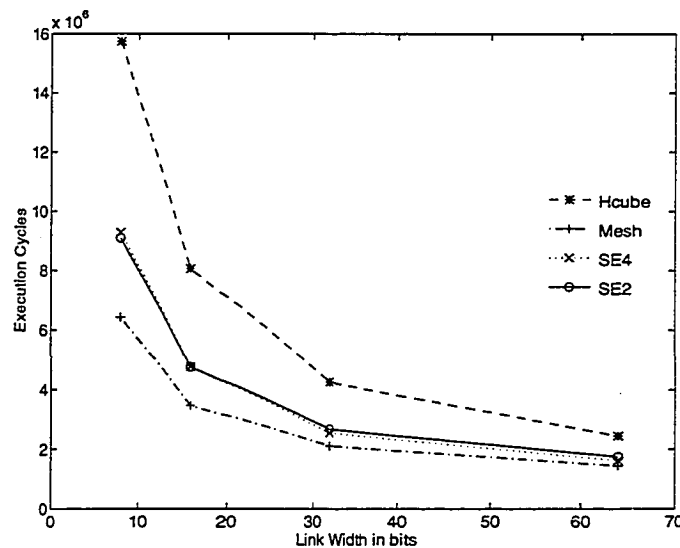


Figure 5.11: Performance of Different Network Topologies with Constant Bisection Width for FFT

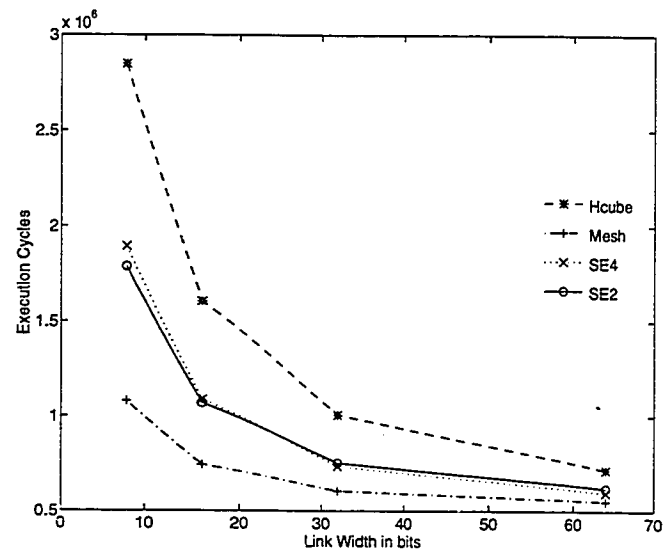


Figure 5.12: Performance of Different Network Topologies with Constant Bisection Width for Bimerge

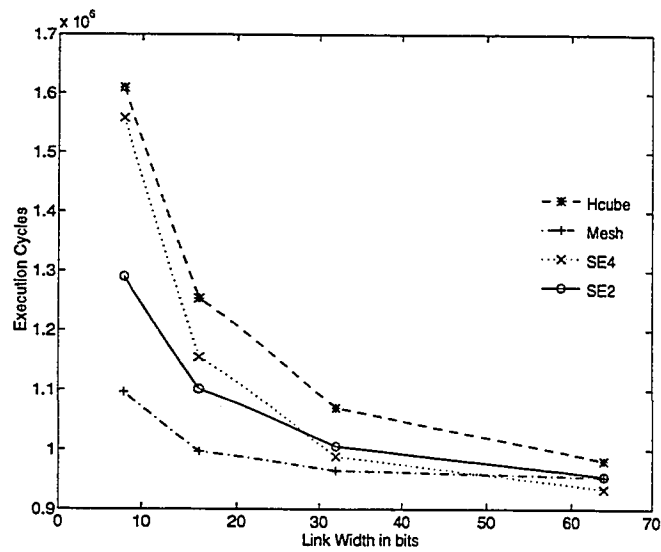


Figure 5.13: Performance of Different Network Topologies with Constant Bisection Width for SOR

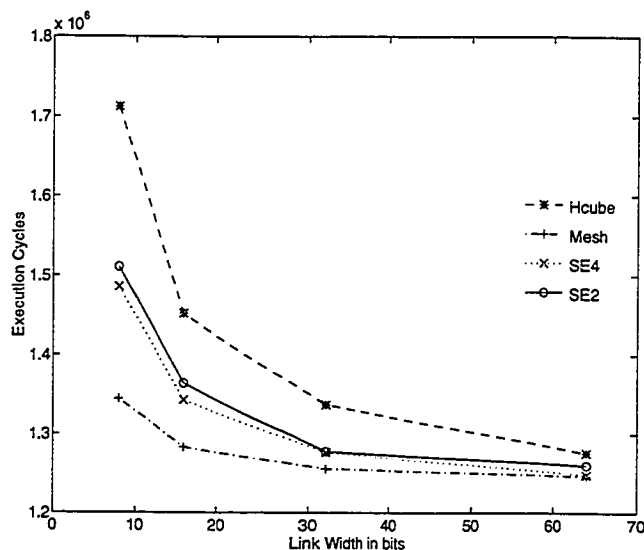


Figure 5.14: Performance of Different Network Topologies with Constant Bisection Width for MMULT

has a link width of 64 bits. This corresponds to the flattening of the curve at higher link widths in the earlier graphs with constant link widths. However, even though the performance gap is closing, the corresponding mesh network still performs better than the corresponding hypercube network. The mesh network will perform better if for the same cost one can build a mesh with wider links than a hypercube.

As the number of processors increases, the bisection width of the hypercube and the shuffle-exchange grows proportional to P , the number of processors. The bisection width of the mesh grows as \sqrt{P} . Therefore for a system with a larger number of processors, the differences in performance among these networks for the same bisection width are going to increase.

5.3 Components of Latency of a Cache Line Fetch

In this section we analyze the various components of the total latency of a cache line fetch and the effect of the network link width on these components. In the previous section, when comparing the performance of different network topologies, we saw that when network cost is taken into consideration the mesh network performs better than the other networks. Therefore in this section we will only analyze message latency for the mesh network. The latency seen at the cache on a miss may include one or more of the following.

- In the case of a remote access, the time to send request through the request network. This includes waiting time in the queue to enter the network, blocking time in the network, and the network latency.
- Time spent in the queue waiting for service at a directory module and time spent accessing directory data structures.
- In case there is a pending access to this line, the time spent in the directory module buffer for this access to complete.
- In case a coherence transaction is needed before this access can complete, the time spent waiting in the directory module while it creates a packet, sends it to the cache module, and awaits a reply.
- In case a transaction needs a memory access, time spent in the queue waiting for service at the memory module and time spent accessing the memory module.
- Time taken to get the reply back to the cache module. This involves another network access in the case of a remote access.
- In case there is no buffer space in the directory module, the time spent re-trying the request.

- In case this request replaced a dirty line in the cache, the time spent in flushing out the dirty line before this access can be forwarded. In our simulation, since we did not use a write buffer, this request is not forwarded until the dirty line has been returned to memory and an acknowledgment is received.

As explained in Section 4.5, the architecture simulated uses a pair of networks in order to avoid deadlocks. One of the networks is used to carry requests and the other is used to carry replies. The request network carries requests for a cache line, coherence requests, and the dirty line on a cache line flush. The reply network carries replies from the directory to the cache and coherence replies from the cache. All packets in the network are one of two sizes. The larger packets carry data back from memory or carry data from cache to memory on a line flush or in response to a coherence request. The size of this packet is 39 bytes: 32 bytes for the data, 4 bytes for the address, and 3 bytes for the request type, source id, and destination id. The small packets carry a request or just an acknowledgment in reply. This packet is 7 bytes long.

The average packet size going through a network varies for each application, depending on the amount of coherence needed for the application. Table 5.4 shows the the number of flits in the large and small packets for the various link widths. This table also shows the latency of the packet when the packet travels one hop. Each additional hop increases the latency by 2 cycles. The latency includes 2 cycles to enter the network and 2 cycles to leave the network. Since the latency is proportional to the sum of the size of the packet and the number of hops, the latency decreases by less than half as the network link width is doubled.

Figure 5.15 shows the sum of average network latency for the request and reply of each cache miss accessing a remote node, for the four applications. This includes the blocking times in the network. Overall the network latencies for the different applications are close to each other. The differences in these curves are due to differences

Link Width	Small Packet		Large Packet	
	Num Flits	Latency	Num Flits	Latency
4	14	32	78	160
8	7	18	39	82
16	4	12	20	44
32	2	8	10	24
64	1	6	5	14

Table 5.4: Latency of the Two Packet Sizes at Various Link Widths

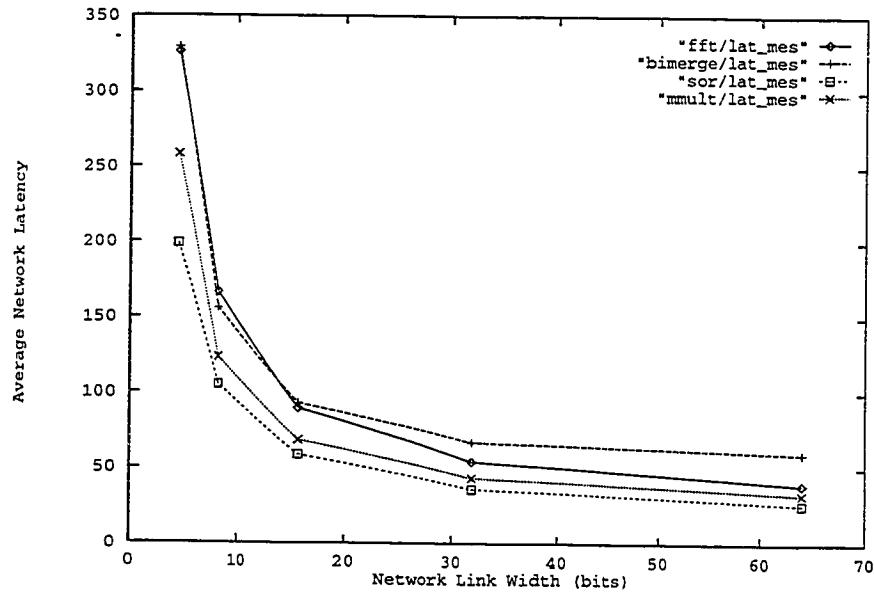


Figure 5.15: Sum of Average Network Latency For Request and Reply in a Mesh Network

in the average number of hops and the differences in blocking time. The change in latency decreases as the link width increases. At lower network link widths, networks are more likely to have contention inside the network. As the link width increases, the network contention and the size of the packet reduces. The latency due to the number of hops becomes a significant fraction of the total latency.

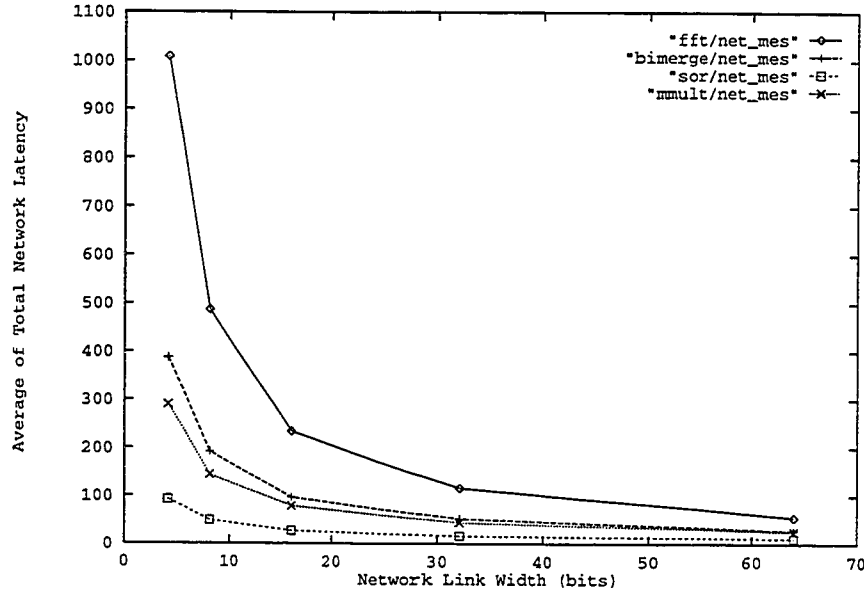


Figure 5.16: Sum of Average Network Latency and Port Waiting Time For Request and Reply in a Mesh Network

Total network time includes the network latency shown in Figure 5.15 and the network port waiting time, which is the time spent by the request waiting to enter the network. The effect of the average network time of a packet on the overall performance of an application, depends on the percentage of packets that access a remote node. Figure 5.16 shows the total network time which accounts for the percentage of requests that are to remote nodes. This is done by multiplying the total network access time (latency + port waiting time) by the fraction of accesses to a remote node. The fraction of accesses to a remote node is 85% in FFT, 44% in Bimerge, 78% in MMULT and 42% in SOR. While the network latency of all applications are close to each other in Figure 5.15, Figure 5.16 shows that the access times are quite disparate at the lower link widths. The FFT has the highest latency and SOR has the lowest latency. This is because the packets in SOR face almost no blocking either at the ports or in the

network. FFT on the other hand has higher blocking times and higher wait times at the port. For FFT at a link width of 4 bits, the port waiting time is almost three times the average network latency. Accesses to local node may also wait at the port queues. The directory puts its replies in a FIFO buffer. When the port buffer is free, the packet at the head of the FIFO is moved to the port queue. A reply to the cache in the same node as the directory is also added to the FIFO and can reach the cache only when it reaches the head of the FIFO.

We discuss the effect of increasing the link widths on the remaining components of the cache miss access:

- We find that the time spent waiting for service at the memory module and the directory module is quite small. But there is a trend in all applications for these waiting times to increase as the link widths are increased. This is because the network becomes less of a bottleneck in the system as the link widths are increased.
- The decrease in the average waiting time for a cache line flush, as the link width increases, depends on the percentage of remote accesses. FFT has the highest percentage of accesses (13%) that have to wait for a cache line flush before sending out the current access.
- The time spent in the directory module for a coherence access to complete depends on the network latency. It also depends on whether the coherence request has to bring back a dirty line and write it to memory. In Bimerge and SOR a large percentage of the accesses (64% and 71% respectively) have to wait for a coherence access to complete. In FFT and MMULT less than 10% of the accesses have to wait for a coherence request.
- When a request has to wait for a conflicting access to complete, the earlier access could be accessing memory or it could be waiting in the directory module queue

waiting for a coherence request to come back. MMULT has mostly read sharing and all processes sharing a data item access it in the same order. Even for this program, the percentage of accesses that have to wait for a conflicting access is quite low at a link width of 4 bits. This value goes up to 10% at a link width of 16 bits and 29% at a link width of 64 bits. At the smaller link widths, the request spends most of its time in the network and it is less likely to conflict in the directory module. All other applications have a low percentage of conflicts at all link widths except 64 bits.

- The time spent to re-try a request because of a full directory buffer is low. Bimerge has the highest percentage of retries (1%). This is because bimerge has a disproportionate number of accesses going to module 0. However, even this value is only 1%, indicating that there is sufficient buffer space allocated in the system. This percentage goes up slightly at a link width of 64 bits.

Figure 5.17 shows the sum of the time spent in all of the above mentioned components for the 4 applications. As seen in the figure, bimerge has a higher memory access latency than the other applications. All processes in this application read and write to a few small arrays, and this conflict causes the high access time. The accesses have to wait at the memory module while a coherence request is sent to the current owner of the cache line. We also see in the figure that all applications show increased memory access latency when the link width is increased from 32 bits to 64 bits. For MMULT the latency starts increasing at 16 bits. When the network access latency decreases, the rate of request at the memory module goes up. Therefore the waiting time in all the queues in the directory and memory sub-system increases. We expect this trend to get worse if the memory module gets slower. Although the FFT curve seems flat, there is a 3-cycle increase in latency for a link width of 64 bits compared with a link width of 32 bits.

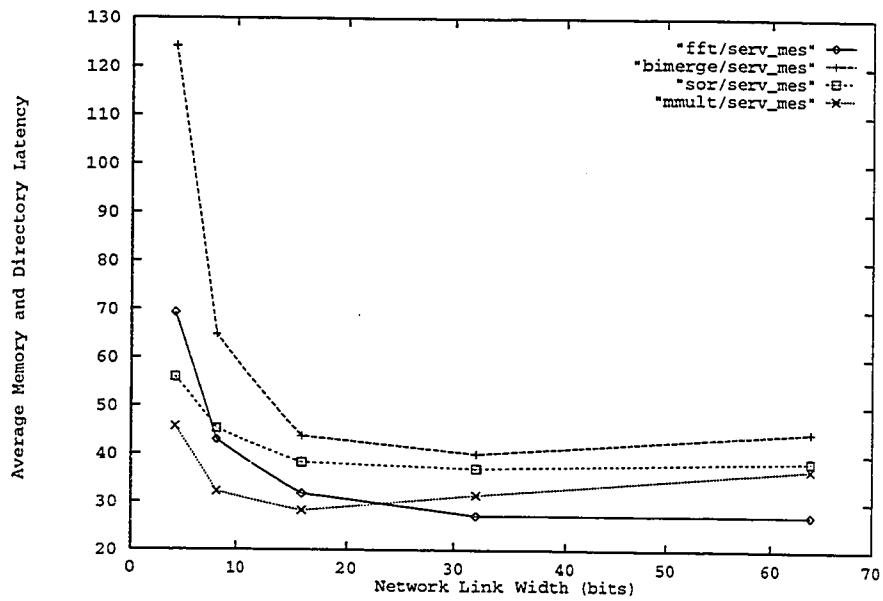


Figure 5.17: Average Service Time and Waiting Time at Directory and Memory Module in a Mesh Network

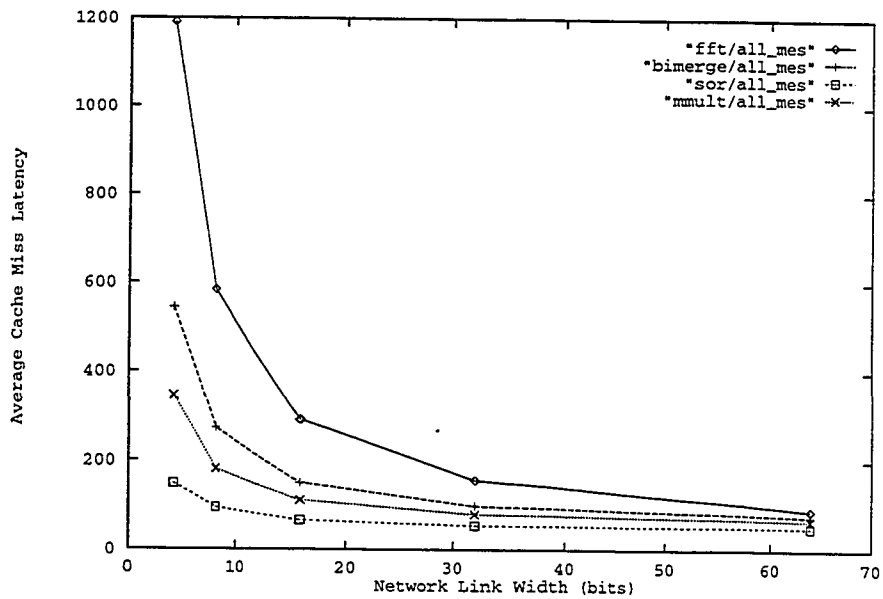


Figure 5.18: Overall Cache Miss Access Latency in a Mesh Network

Link Width	FFT (%)	BIMERGE (%)	SOR (%)	MMULT (%)
8 / 4	103.90	99.12	57.06	90.97
16 / 8	100.58	82.74	43.51	63.44
32 / 16	86.43	52.27	21.46	39.13
64 / 32	75.13	29.07	6.14	18.84

Table 5.5: Percentage Improvement in Cache Miss Latency with a Mesh Network When Link Width is Increased

Link Width	FFT (%)	BIMERGE (%)	SOR (%)	MMULT (%)
8 / 4	96.63	69.24	15.29	10.46
16 / 8	85.56	45.51	10.02	4.80
32 / 16	64.05	22.84	3.31	2.17
64 / 32	46.06	9.68	1.12	0.76

Table 5.6: Percentage Improvement in Performance with a Mesh Network When Link Width is Increased

Figure 5.18 shows the latency seen at the cache module on a miss. All the curves in this figure are obtained by a sum of the corresponding points in Figures 5.16 and 5.17. Table 5.5 gives the percentage improvement for this latency as the link width is increased. Table 5.6 gives the percentage improvement in the overall execution times. As seen in Table 5.5, FFT has the most improvement in latency when the link width is increased. This is followed by Bimerge, MMULT and then SOR. Table 5.6 shows that SOR has a greater overall performance improvement than MMULT when

the link widths are increased. Also, the percentage improvements seen for the cache latency are not reflected in the overall performance. The improvement in the overall performance depends on both the cache hit rate and the cache miss latency.

Cache miss rate is calculated as the ratio of the number of accesses that cause a cache miss to the total number of accesses to the cache. However, because the processes spin on the lock variable while another process owns the lock, the total number of accesses to the cache is not entirely program-dependent. When the network has a smaller link width and hence longer network access latencies, it takes longer to propagate the cache line invalidates caused by unlock. The processes spin for longer time and cause total number of accesses to be higher. These accesses almost always hit in the cache and the hit rate in such cases will seem high. In order to determine the cache miss rate independent of the network architecture, we ran a simulation with 0 network latency and 1 cycle to access the memory module and directory module.

From our experiment we determined that when running FFT, caches have an average hit rate of 84%. The hit rate for BIMERGE is 96.7%, the hit rate for SOR is 94.9% and the hit rate for MMULT is 99.3%. Since MMULT has the very high hit rate, its overall performance is not as affected by changes in cache miss latency. FFT which has the lowest cache hit rate is most affected by the cache miss latency and hence the changes in link width.

In conclusion

- The link width of 4 bits does not perform well. All applications show a significant improvement in performance when the link width is increased to 8 bits.
- There does not seem to be a case to use 64-bit wide links in a system with a 32-byte cache line. With a link width of 64 bits the network is no longer the bottleneck. The requests spend more time waiting in queues in directory or memory modules than with the network with 32-bit wide links. There is little net gain in performance when going from 32 bits to 64 bits. The performance

improvement of 64-bit links over 32-bit links is greater than 10% only for one application.

- With 8-bit wide links, there is a significant amount of blocking in the network and at the ports for some applications. FFT and Bimerge show an 85% and a 45% improvement respectively, and SOR shows a 10% improvement when the link width is increased from 8 bits to 16 bits.
- The choice between 16-bit wide links and 32-bit wide links is not as clear. For two out of the four applications (SOR and MMULT), the performance improvement in going to 32-bit links from 16 is less than 5%, while the other two applications have a 22% and a 64% improvement. When we look at the increase in time spent in the memory sub-system (see Figure 5.17), it is again split along the same lines. For two applications, this waiting time goes up when we increase link width from 8 bits to 16 bits. For the other two applications, this value does not start increasing until the link width is 32 bits. We need to study more applications before any conclusion can be made about these two link widths.
- A weakly consistent system will increase the load in the system, in which case a larger network bandwidth will be more useful. On the other hand, if the relative speed of memory with respect to the processor is lower, then lower bandwidths are adequate.

5.4 Effect of Memory Speeds on Performance

In the previous section, we saw that at a link width of 64 bits the performance of the multiprocessor is limited by the memory sub-system rather than the network. This motivated us to investigate what happens if the memory subsystem gets slower with respect to the network. In the previous section, we used a processor speed

of 50 MHz and a memory with an access time of 60 ns. With 64-bit wide data paths inside a node, this meant a cache line of 32 bytes could be read in 10 cycles. While 60 ns DRAMs are available in current workstations, the processor speeds in current workstations are around 100 MHz. There are also some 150 MHz processors in workstations using the same speed DRAMs. We ran two sets of simulations, one with 20 cycles latency to fetch a cache line from memory and the other with 40 cycle latency. We kept the proportion of network to processor speeds the same as before (2:1). The directory access time was also increased since the directory module is also implemented in DRAM technology.

Figures 5.19 and 5.20 show the execution times for the three memory speeds for the mesh network for FFT and SOR, respectively. FFT does not show significant differences in performance for the different memory speeds at link widths of 4 and

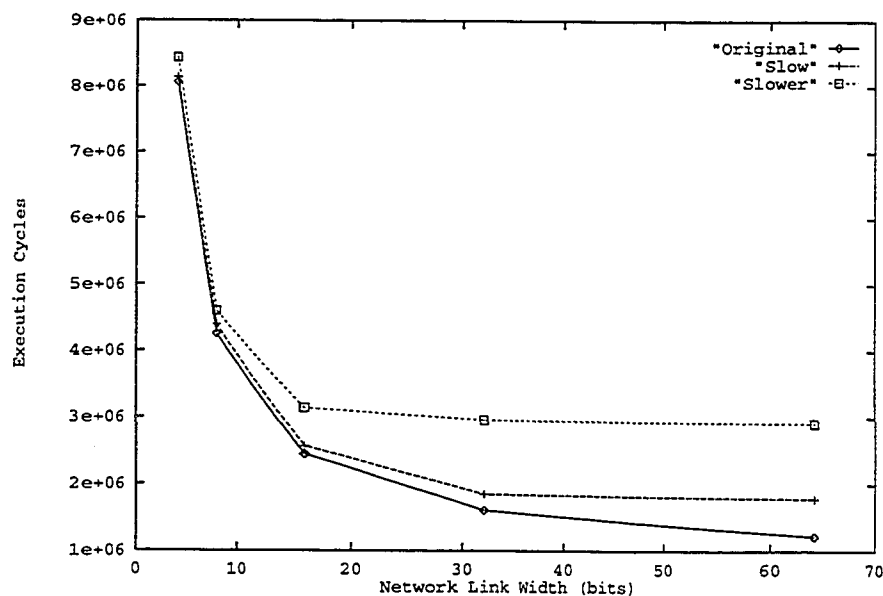


Figure 5.19: Performance Difference in FFT as the Memory Speeds are Increased

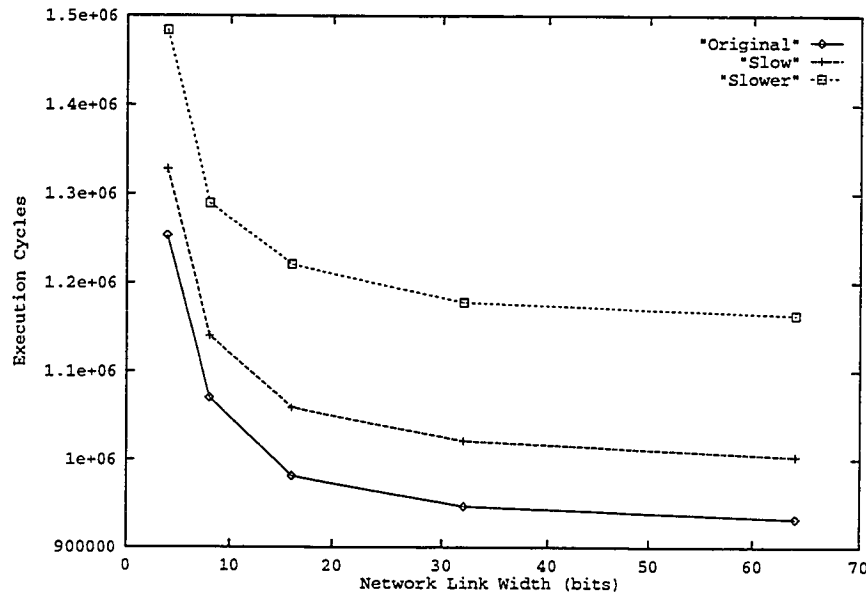


Figure 5.20: Performance Difference in SOR as the Memory Speeds are Increased

8 bits, whereas for SOR the performance difference is substantial at all link widths. MMULT and Bimerge have performance differences which are between these two extremes.

Comparing the performance with slower memory with the original performance, we find that as long there is a significant blocking time in the network and the ports, this time is redistributed when the memory is slower. The network blocking time decreases and the requests spend more time waiting or in service at the memory module. Therefore there is no change in performance with slower memory. However, if the blocking time is not too high, then the slower memory affects the overall performance. Not only is the memory service time longer, but the time waiting to get memory service also increases. The corresponding percentages of re-tries and requests

waiting due to access conflicts also go up. As a result the overall execution time is higher.

Table 5.7 shows the relative improvement in performance as the link width is increased, for the four applications and the two slower memory speeds. When the link width is increased to 16 bits from 8 bits, FFT and Bimerge see significant improvements in performance at both of the slower memory speeds. As the link width is increased to 32 bits, only FFT still has significant performance improvements. Therefore if the memory sub-system is slower the architecture may be able to make the best use of a network with 16-bit wide links.

5.5 NETSIM Simulation Complexity

In this section we report the change in execution time predicted by the simulator, when the network-related contention is not modeled. We also discuss our results on the overhead involved in the detailed simulation of networks when simulating a shared-memory multiprocessor. In the design of a parallel processor, several components in the system have to be evaluated to design a system with an optimal configuration.

Link Width	FFT		BIMERGE		SOR		MMULT	
	Slw	Slwr	Slw	Slwr	Slw	Slwr	Slw	Slwr
8 / 4	93.6	90.2	65.5	51.9	16.0	14.1	9.6	9.7
16 / 8	81.4	82.6	37.5	19.5	7.7	6.4	4.7	4.0
32 / 16	61.0	18.6	13.5	2.9	4.6	3.1	1.8	1.2
64 / 32	18.3	2.8	2.0	1.0	1.4	1.5	0.6	0.7

Table 5.7: Percentage Improvement in Performance as the Link Widths Are Increased. (2 Memory Speeds)

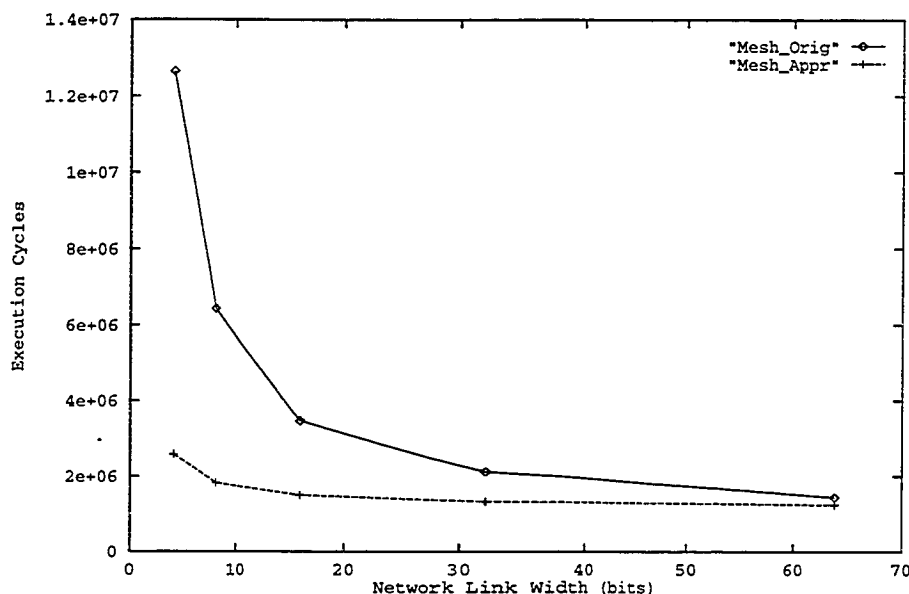


Figure 5.21: Execution Time Predicted by Detailed Network Simulation and Approximate Network Simulation for FFT

Simulation is often used to evaluate the tradeoffs in the design of these components. There have been many experiments evaluating the performance of shared-memory multiprocessors that ignore the contention in the network ([44, 22, 21]). Most of this work concentrated on increasing processor utilization by either tolerating latency or reducing the latency of remote fetches. Latency hiding techniques such as context switching to another thread while waiting on a remote access or weaker consistency models that allow multiple accesses to be outstanding, increase the utilization of the network and could potentially lead to more contention in the network. When studying one such new technique, it is important to understand its effect on the system. In order to keep the analysis simple it might be necessary to initially run these experiments without simulating network contention. However, eventually the network contention must be simulated, since the potential performance of a system

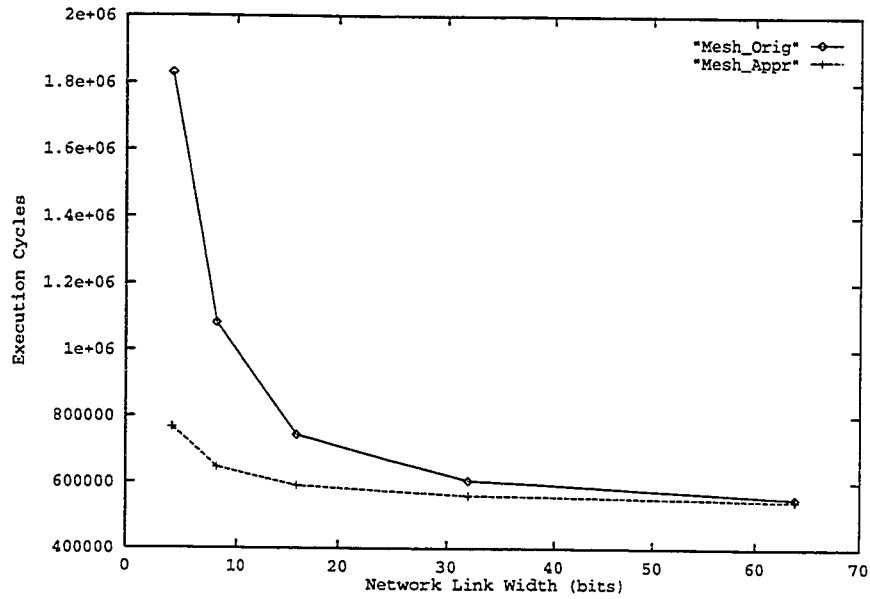


Figure 5.22: Execution Time Predicted by Detailed Network Simulation and Approximate Network Simulation for Bimerge

could never be realized if the network performance does not match the rest of the system.

In seeing the results presented so far, one can see that there is a significant amount of blocking time in the network, especially at smaller network link widths. Nevertheless, we ran a set of experiments that do not simulate the network in detail. In these experiments all the directory and cache modules were connected to each other by a full crossbar. In order to send a message from a cache to a specific directory or memory module, the simulator creates a YACSIM event for that packet. It delays for the latency of the packet and at the end of that time adds the packet to the port of the directory module. The latency is calculated as follows. Given the source and destination address of the packet, the simulation calculates the number of hops the message would have taken in the target network. Using this number of hops and the

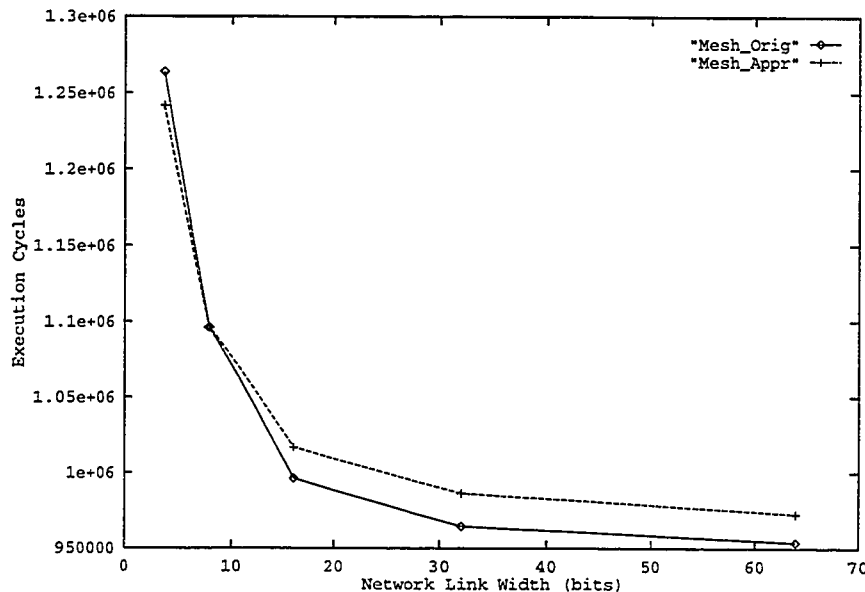


Figure 5.23: Execution Time Predicted by Detailed Network Simulation and Approximate Network Simulation for SOR

size of the packet, it calculates the latency of the packet for a given bandwidth of the network. By contrast, when the network is simulated in detail using NETSIM, an event is associated with the head and tail of the packet. These events are rescheduled at every buffer along the route in the network.

Figures 5.21 to 5.24 show the execution times predicted by the detailed simulation using NETSIM and approximate simulation at various link widths of the mesh network. Since the approximate simulation does not model contention, the applications that have a high contention also show a significant difference in performance. For FFT there is a 79% difference in execution times at a link width of 4 bits. This goes down to 16% at a link width of 64 bits. The difference in execution times is a little closer in Bimerge than in FFT. The difference in this case starts at 58% and goes

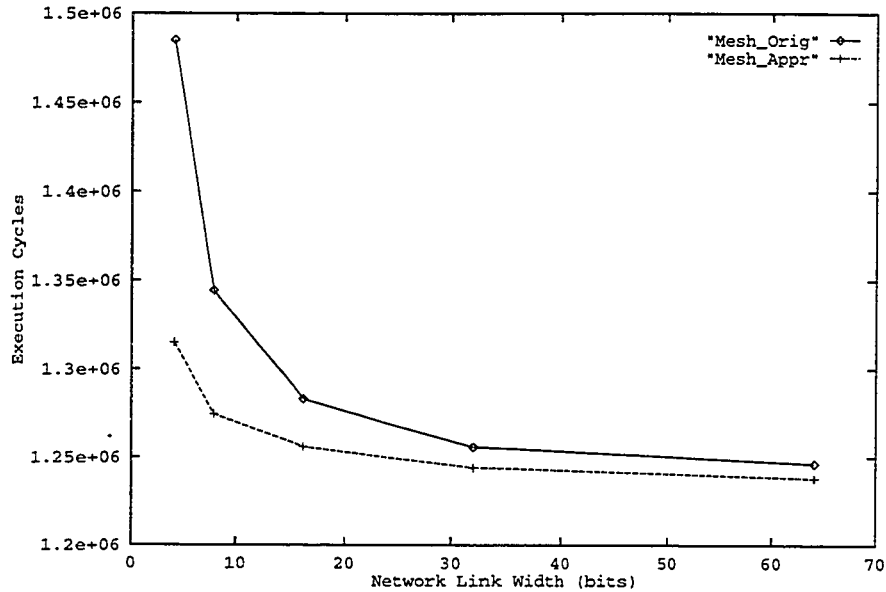


Figure 5.24: Execution Time Predicted by Detailed Network Simulation and Approximate Network Simulation for MMULT

down to 1%. MMULT follows Bimerge with a difference in execution time varying between 11% and 0.67%.

It is interesting that the approximate simulation predicts a higher execution time than the detailed simulation for SOR. Examining the results, we find that the average cache latency is lower in the approximate simulation, but the average number of accesses is higher. The directory module accesses the data in a FCFS basis for the detailed simulation. In the approximate model the requests are added to individual ports and the directory services requests in a round-robin fashion. We believe that these differences in request service pattern at the directory module change the overall execution time somewhat. The key point to observe here is that the difference in performance between the two simulations is only $\pm 2\%$, because SOR has very little blocking time in the network.

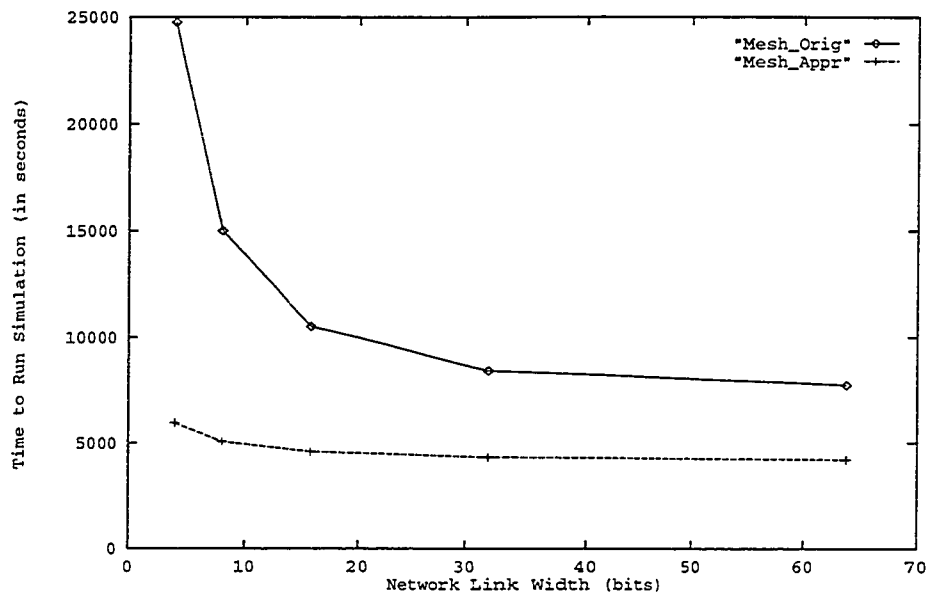


Figure 5.25: Simulation Time Taken by Detailed Network Simulation and Approximate Network Simulation for FFT

Figures 5.25 to 5.28 show the time taken to run the simulations. The simulation times are measured by using the UNIX *time* command. The simulations were not run under controlled conditions and therefore the absolute values are questionable. However, the general trend that is observed will hold even if the simulations are run under controlled conditions. As can be seen in these figures, the approximate simulation takes about just as long to run simulations of all link widths. This is because the time to simulate a packet is independent of the number of flits in the packet. For Bimerge and FFT it takes slightly longer to run the simulation at a link width of 4 bits. At these link widths, the longer latencies cause some processors to spin wait longer for synchronization. Each access to the cache causes a trap from the simulator, and these cases take slightly longer to run.

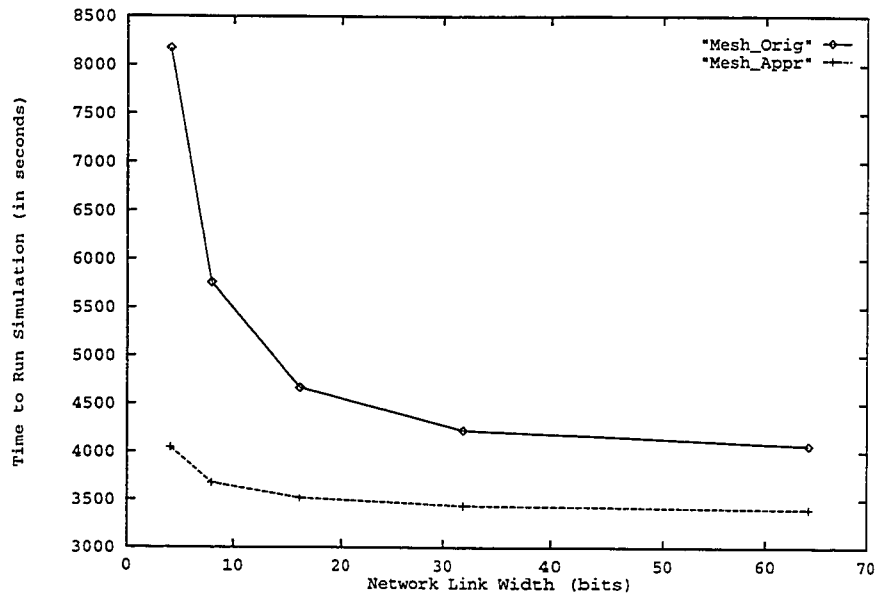


Figure 5.26: Simulation Time Taken by Detailed Network Simulation and Approximate Network Simulation for Bimerge

In the case of the detailed simulation, the lower link widths always take longer to run than the higher link widths regardless of the amount of blocking, since there is a greater number of flits in the packet at the smaller link widths. The tail event, which manages the distribution of flits between the head flit and tail flit, has more flits to manage. In addition to this, NETSIM takes longer when there is blocking in the network. However, all the differences in time between the two simulations cannot be attributed to NETSIM. When the directory module is ready to send out a reply and the port buffer is full, the reply is queued in the directory buffer space. When the port buffer later can accept the reply, the event associated with the directory module is woken up and it transfers the reply to the port buffer. This potentially repeated re-scheduling of the directory module is avoided in the approximate simulation.

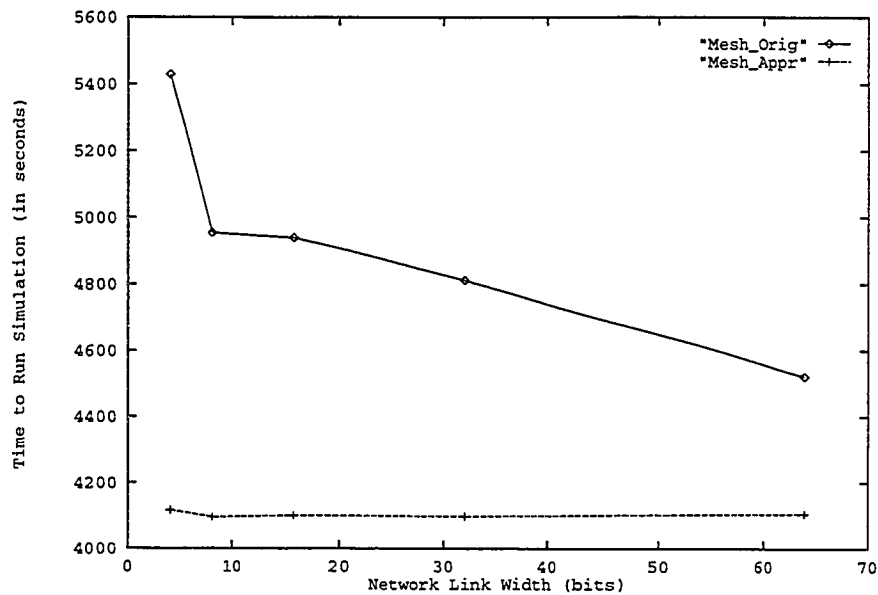


Figure 5.27: Simulation Time Taken by Detailed Network Simulation and Approximate Network Simulation for SOR

In conclusion, the performance difference can be significant in some cases if the network contention is ignored. We found a difference in execution times of up to 78% depending on the network parameters and the application considered.

The difference in time to run the two types of simulations can also be significant. We found this difference to vary between 75% and 8% depending on the network parameters and the application considered. In the case of SOR this difference is relatively low, varying between 9% and 25%. Significant savings in time are only achieved when there is a lot of blocking in the network. The detailed network simulation is quite efficient when there is no contention in the network.

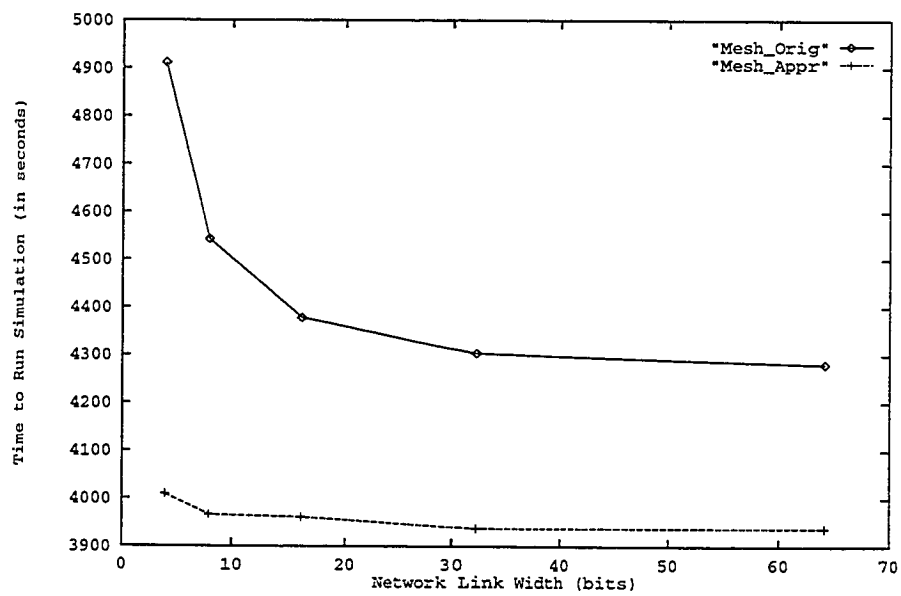


Figure 5.28: Simulation Time Taken by Detailed Network Simulation and Approximate Network Simulation for MMULT

Chapter 6

Conclusions

This thesis has examined the effect of the choice of interconnection networks on the performance of shared-memory multiprocessors. The network performance was studied using execution-driven simulation. The performance of four network topologies - mesh, hypercube and two shuffle-exchange configurations - were studied. The performance of each network topology was studied at different link widths varying between 4 bits and 64 bits. Four applications were used in our experiments: Fast Fourier Transform (FFT), Bimerge (a version of merge-sort), Matrix Multiply (MMULT) and Successive Over Relaxation (SOR).

The main results of our study can be summarized as follows:

- With constant bisection width, the mesh network outperforms the other network topologies
- Cache miss rate largely influences the relative performance of different network configurations
- When memory speeds relative to the processor speed is reduced, the network bandwidth available can also be reduced without significant loss in performance
- Time taken for detailed network simulation is proportional to the amount of contention in the network
- Performance results obtained with the approximate network simulation (assuming zero contention in the network) can be significantly different from the

results obtained with detailed network simulation (up to 80% difference in our experiments)

6.1 Conclusions on Network Topology Experiments

When comparing the different network topologies with constant link widths, we find that the difference in performance depends on the difference in the average blocking time in the port and in the network. The average number of hops taken by a packet does not affect the overall performance significantly. This indicates that the bandwidth of the network is critical for performance, as expected with wormhole routing.

When comparing networks with 64 nodes and constant link widths, the hypercube outperforms all the other networks and the shuffle-exchange outperforms the mesh. The mesh network performs comparatively poorly for FFT and Bimerge. Both of these applications have non-local communication. In the case of FFT, all processors access all nodes in the machine heavily (if not equally) in order to fetch some pre-computed sine values. In each iteration of Bimerge, every processor updates its respective entry in a small array, which is stored in node 0. Although the average number of hops in the mesh network is not much higher than in the hypercube, even for such global communication patterns, the mesh network pays a penalty in higher blocking time. For both of these problems, the mesh network has higher network blocking time and higher port blocking time.

The performance of the shuffle-exchange is interesting, especially when the performance of the two configurations, one with a switch size of 2 and the other with a switch size of 4, are compared. Selecting the switch size of a multistage network involves a tradeoff between the contention at the switch and the latency involved in having multiple stages. For wormhole routed networks, the impact of latency on performance is low. When there is no blocking SE4 does slightly better than SE2.

When there is contention at the switch, sometimes the traffic pattern is such that both networks have about the same amount of blocking time. However, when there is nearest neighbor communication as in SOR, the SE4 does worse. In this case all input links try to use the same output link. In SE4 four nodes are contending for a link, whereas in SE2 only two nodes are contending for a link. As a result, for lower link widths SE4 does worse than all the other networks for SOR. When the networks with equal bisection width are compared, the mesh far outperforms the other networks.

6.2 Conclusions on Comparing Different Link Widths

The performance difference is insignificant for most applications when the link width is increased from 32 bits to 64 bits. The FFT is the only application to see a significant performance gain (46%) when the link width is increased from 32 bits to 64 bits. Networks with link widths of 4 bits and 8 bits experience significant blocking. While link width of 16 bits and 32 bits appear to offer a reasonable tradeoff between performance and cost, the choice between these two bandwidths is not clear.

The cache hit rate is one of the factors that determines the load on the network. The relative performance of different network bandwidths for a particular application depends on the cache hit rate for the application, the fraction of the miss accesses that are to remote nodes, the average distance traveled in the network by a packet, and the average size of these packets. Among these factors, cache hit rate is the most significant. For the applications with higher cache hit rate (above 90%), the differences in performance is insignificant when the link width is increased from 32 bits to 64 bits. The FFT has the lowest hit rate (84%) and has the most significant performance gain when the link width is increased from 32 bits to 64 bits. The hit rate of the application is very important in architectures such as this with large latencies on a cache miss.

We also studied the relative performance of each network topology at different link widths when the memory sub-system is slower. We ran one set of experiments with the memory twice as slow as before(*Slow*) and one set of experiments with the memory four times as slow as before(*Slower*). With a *Slow* memory the tradeoffs are still somewhat similar to those for the original memory, although in this case one would probably choose a 16-bit wide network over the 32-bit wide network. With the *Slower* memory, it is clearly sufficient to use a 16-bit wide links in the network.

6.3 Conclusions on Network Complexity

In order to avoid detailed simulation of the network, the traversal of each packet through the network can be simulated by delaying for the minimum latency for this packet to traverse the network, calculated from the distance between source and destination nodes and the size of the packet. This does not model the contention in the network and waiting time at the ports. When results from a run of the applications using the approximate simulation are compared to the detailed simulation results, we find that they are significantly different (up to 80%) when the blocking time in the network and port is high. For SOR the difference is low at almost all link widths, since the network packets in SOR do not have high contention due to the nearest neighbor communication pattern. For the other applications, the results do not become similar until we reach link widths of 32 and 64 bits.

When the times to run the detailed and approximate simulations are compared, there is a difference ranging from 8% to 75% depending on the network parameters and the applications considered. While this might seem like a significant improvement, the performance difference can also be quite significant, making it inadvisable to simulate the system without the detailed network simulator.

6.4 Future Work

The experiments in this thesis were conducted for four applications chosen from numeric and non-numeric domains of study. While these applications are representative of the workload in multiprocessors, it is also desirable to run the experiments on more applications and confirm the results observed in this thesis.

In all applications that we studied the size of the active data set in any phase during execution was smaller than the cache size, resulting in cold start, conflict, and coherence misses, but not capacity misses. The results presented here study the performance of different network parameters due to traffic from cold start, coherence and conflict misses. It is desirable to determine the effect of capacity misses on the relative performance of different network parameters. The problem sizes and the cache sizes were scaled down in our experiments, in order to complete the large number of simulations in a timely manner. Although the traffic pattern of an application does not change with problem sizes, it is still desirable to run these experiments with more realistic problem sizes in order to obtain a greater level of confidence in our results.

Three network topologies, hypercube, mesh and shuffle-exchange, were chosen for two reasons: they are substantially different, and they have all been used in commercial multiprocessors. A fat-tree is quite different in topology from these networks and is used in the CM-5 MIMD machine. It will be interesting to see how the fat-tree fares in comparison to the other networks. The effect of other cache line sizes should be studied, since the size of the network packet and the cache hit rate depend on the cache line size.

All the experiments in this thesis were on systems with sequential consistency. With such a consistency model, at most a processor can have a single access outstanding. When the consistency is weakened, each node can proceed after issuing a write by buffering the data and writing to memory when possible. Therefore there can be multiple outstanding writes in a node at a time. This increases the processor

utilization. It also increases the load on the network and the network performance under such circumstances deserves more investigation.

The network itself was studied with wormhole routing and deterministic routing, where the routing does not take into account the current state of the network. Several different adaptive routing schemes that use virtual channels to avoid deadlock have been proposed in the literature. Adaptive routing can potentially make better use of the network. It remains to be seen whether significant gains can be achieved from using an adaptive routing protocol to overcome the cost and complexity of implementation. Currently the architecture simulated uses a pair of networks in order to avoid deadlock between directories. With the use of virtual channels, the two networks can be multiplexed onto a single network. The performance of such an architecture must also be studied, since the use of two networks, where one of them is highly under-utilized (the request network carrying mostly smaller packets) seems wasteful.

Bibliography

- [1] S. Abraham and K. Padmanabhan. Performance of the direct binary n-cube network for multiprocessors. *IEEE Transactions on Computers*, 38(7):1000–1011, July 1989.
- [2] A. Agarwal. Limits on interconnection network performance. *IEEE Transactions on Parallel and Distributed Systems*, 2(4):398–412, October 1991.
- [3] A. Agarwal, B. Lim, D. Kranz, and J. Kubiawicz. APRIL: A processor architecture for multiprocessing. In *Proceedings of the 17th International Symposium on Computer Architecture*, pages 104–114, May 1990.
- [4] D.H. Bailey, E. Barszcz, L. Dagum, and H.D. Simon. NAS parallel benchmark results. In *Supercomputing, '92*, pages 386–393, 1992.
- [5] E.A. Brewer, C.N. Dellarocas, A. Colbrook, and W.E. Weihl. PROTEUS: A high-performance parallel-architecture simulator. Technical Report MIT/LCS/TR-516, Massachusetts Institute of Technology, September 1991.
- [6] D. Chaiken, C. Fields, K. Kurihara, and A. Agarwal. Directory-based cache coherence in large-scale multiprocessors. *Computer*, 23(6):49–58, June 1990.
- [7] S. Chittor and R. Enbody. Performance evaluation of mesh-connected wormhole-routed networks for interprocessor communication in multicomputers. In *Proceedings of Supercomputing '90*, pages 647–656, 1990.
- [8] Intel Scientific Computers. *iPSC2*. 1988.

- [9] Intel Scientific Computers. *The Paragon XP/S Architecture*. 1992.
- [10] Encore Computer Corporation. *Multimax technical summary*. 1990.
- [11] Ncube Corporation. *The nCUBE2 technical reference manual*. 1988.
- [12] Thinking Machines Corporation. *The Connection Machine CM-5 technical summary*. 1991.
- [13] W.J. Dally. Performance analysis of k-ary n-cube interconnection networks. *IEEE Transactions on Computers*, 39(6):31–58, June 1990.
- [14] W.J. Dally and C.L. Seitz. Deadlock-free message routing in multiprocessor interconnection networks. *IEEE Transactions on Computers*, C-36(5):547–553, May 1987.
- [15] H. Davis, S.R. Goldschmidt, and J. Hennessy. Tango: A multiprocessor simulation and tracing system. Technical Report CSL-TR-90-4399, Computer Systems Laboratory, Stanford University, 1990.
- [16] D.M. Dias and J.R. Jump. Analysis and simulation of buffered delta networks. *IEEE Transactions on Computers*, C-30(4):273–282, April 1981.
- [17] S. Dwarkadas, J.R. Jump, and J.B. Sinclair. Efficient simulation of cache memories. In *1989 Winter Simulation Conference Proceedings*, pages 1032–1041, December 1989.
- [18] T.-Y. Feng. A survey of interconnection networks. *Computer*, 14(12):12–27, December 1981.
- [19] D. Gajski, D. Kuck, D. Lawrie, and A. Saleh. Cedar - A large scale multiprocessor. In *Proceedings of the International Conference on Parallel Processing*, pages 524–529, 1983.

- [20] K. Gharachorloo, A. Gupta, and J. Hennessy. Performance evaluation of memory consistency models for shared-memory multiprocessors. In *ASPLOS-IV*, pages 245–257, 1991.
- [21] K. Gharachorloo, A. Gupta, and J. Hennessy. Hiding memory latency using dynamic scheduling in shared-memory multiprocessors. In *ASPLOS-V*, pages 22–33, 1992.
- [22] A. Gupta, J. Hennessy, K. Gharachorloo, T. Mowry, and W.-D. Weber. Comparative evaluation of latency reducing and tolerating techniques. In *18th Annual International Symposium on Computer Architecture*, pages 254–263, 1991.
- [23] A. Gupta, W.-D. Weber, and T. Mowry. Reducing memory and traffic requirements for scalable directory-based cache coherence schemes. In *International Conference on Parallel Processing*, pages 312–321, 1990.
- [24] D. Hensgen, R. Finkel, and U. Manber. Two algorithms for barrier synchronization. *International Journal of Parallel Programming*, 17(1):1–17, 1982.
- [25] W.D. Hillis. *The Connection Machine*. MIT Press, Cambridge, MA, 1985.
- [26] J.R. Jump. *YACSIM Reference Manual*. Electrical and Computer Engineering Department, Rice University, 1992.
- [27] J.R. Jump. *NETSIM Reference Manual. Version 1.0*. Electrical and Computer Engineering Department, Rice University, 1993.
- [28] J.R. Jump and S. Lakshmanamurthy. NETSIM - A general-purpose interconnection network simulator. In *MASCOTS'93*, pages 121–125, 1993.
- [29] S.D. Kaushik *et al.* An algebraic theory for modeling direct interconnection networks. In *Supercomputing, '92*, pages 488–497, 1992.

- [30] P. Kermani and L. Kleinrock. Virtual cut-through: A new computer communication switching technique. In *Computer Networks*, pages 267–286, 1979.
- [31] R.E. Kessler and J.L. Schwarzmeier. CrayT3D: A new dimension for Cray Research. In *COMPCON*, pages 176–182, 1993.
- [32] D. Lenoski, J. Laudon, K. Gharachorloo, W.-D. Weber, A. Gupta, J. Hennessy, M. Horowitz, and M.S. Lam. The Stanford DASH multiprocessor. *Computer*, 25(3):63–79, March 1992.
- [33] T. Lovett and S. Thakkar. The Symmetry multiprocessor system. In *Proceedings of the 1988 International Conference on Parallel Processing*, pages 303–310, August 1988.
- [34] K. Nakayama. New discrete fourier transform algorithm using butterfly structure fast convolution. *IEEE Transactions on Acoustics, Speech, and Signal Processing*, 33(5):1197–1208, October 1985.
- [35] L.M. Ni and P.K. McKinley. A survey of wormhole routing techniques in direct networks. *Computer*, 26(2):62–76, February 1993.
- [36] G.F. Pfister and V.A. Norton. “Hot Spot” contention and combining in multi-stage interconnection networks. In *Proceedings of the 1985 International Conference on Parallel Processing*, pages 943–948, 1985.
- [37] R. Ponnusamy, A. Choudhary, and G. Fox. Communications overhead on the CM5: An experimental performance evaluation. In *Frontiers, '92*, pages 108–115, 1992.
- [38] W.H. Press, B.P. Flannery, S.A. Teukolsky, and W.T. Vetterling. *Numerical Recipes in C*. Cambridge University Press, Cambridge, 1988.

- [39] S.K. Reinhardt, S.D. Hill, J.R. Larus, A.R. Lebeck, J.C. Lewis, and D.A. Wood. The Wisconsin Wind Tunnel: Virtual prototyping of parallel computers. In *Proceedings of the 1993 ACM SIGMETRICS Conference on Measuring and Modeling of Computer Systems*, May 1993.
- [40] C.L. Seitz. The Cosmic Cube. *Communications of the ACM*, 28(1):22–33, January 1985.
- [41] A.J. Smith. Design of CPU cache memories. In *Proceedings of IEEE TEN-CON'87, Region 10 Conference*, pages 1–10, 1987.
- [42] J.E. Smith and J.R. Goodman. A study of instruction cache organizations and replacement policies. In *10th Annual International Symposium on Computer Architecture*, pages 132–137, 1983.
- [43] P.J. Varman, B.R. Iyer, D.J. Haderle, and S.M. Dunn. Parallel merging: Algorithm and implementation results. *Parallel Computing*, 15(1):165–177, 1990.
- [44] W.-D. Weber and A. Gupta. Exploring the benefits of multiple hardware contexts in a multiprocessor architecture: Preliminary results. In *16th Annual International Symposium on Computer Architecture*, pages 273–280, 1989.
- [45] M.-l. Woo and R.A. Renaut. Parallel power-of-two FFTs on hypercubes. In *Supercomputing, '91*, pages 754–763, 1991.