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.
Efficient simulation of interconnection networks

Lakshmanamurthy, Sridhar, M.S.
Rice University, 1993
RICE UNIVERSITY

Efficient Simulation of Interconnection Networks

by

Sridhar Lakshmanamurthy

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

Master of Science

APPROVED, THESIS COMMITTEE:

Dr. J. R. Jump, Chairman
Professor
Electrical and Computer Engineering

J. B. Sinclair
Dr. J. B. Sinclair, Associate Professor
Electrical and Computer Engineering

Dr. P. J. Varman, Associate Professor
Electrical and Computer Engineering

Houston, Texas
April, 1993
ABSTRACT

Efficient Simulation of Interconnection Networks

by

Sridhar Lakshmanamurthy

This thesis presents a modular, general-purpose interconnection network simulator called NETSIM developed as an extension to YACSIM, a C-language based process-oriented discrete event simulator. NETSIM is capable of simulating a wide range of switch architectures and network topologies. A NETSIM model can be simulated in a stand-alone mode with the network traffic generated by stochastic processes, or it can be used as an integral part of PARCSIM in an execution-driven simulation environment. This work emphasizes simulating complex switching schemes like wormhole routing efficiently. A detailed model for wormhole routing is implemented using two different techniques, and their relative advantages are evaluated. The thesis also presents four approximate models for simulating wormhole routing and evaluates the tradeoff between speedup and the accuracy of the approximate models. This thesis also develops a validation model for the iPSC/860 interconnection network using the NETSIM primitives, and evaluates the accuracy of the model in predicting the message latencies for the iPSC/860 system.
To Amma and Appa...
Acknowledgments

I would like to express my gratitude to Dr. Jump for providing constant guidance during this work. He was always available for comments and suggestions, and his keen insight helped me look at issues in the proper perspective. I would also like to thank Dr. Sinclair and Dr. Varman for their valuable suggestions during the course of this research, and for serving on my thesis committee.

I would like to thank Dr. Pierce of Intel Corporation for answering several of my queries about the iPSC/860 that helped me develop an accurate validation model, and Dr. Driskill of the English Department at Rice for her feedback on my writing style.

I would also like to thank all my friends and colleagues for making this work enjoyable. A few of them deserve special mention - Roomkey for his suggestions during this work, my other officemates, Kotax and Ψχο for helping me with \LaTeX{} and my roommate, Sundeep for spreading the cheer around in the apartment. I would also like to thank Phyllis and Ted Ayre, and Cleora Cummers for their love and support. Special thanks to Anindita and Rajat Mukherjee for making my life here at Rice so much more exciting. And finally, I am eternally indebted to my parents, without whose constant love, support and blessings, over the years, none of this would have been possible.
Contents

Abstract ii
Acknowledgments iv
List of Illustrations ix

1 Introduction 1

1.1 Motivation and Objective .................................. 1
1.2 Related Work .............................................. 3
1.3 Overview of the Thesis .................................... 4

2 Interconnection Networks 5

2.1 Topology .................................................... 6
2.1.1 Direct Networks ......................................... 7
2.1.2 Indirect Networks ....................................... 8
2.2 Routing Protocol and Flow Control ........................ 9
2.3 Switching Strategies ...................................... 11
2.3.1 Store and forward routing .............................. 12
2.3.2 Wormhole routing ...................................... 12
2.3.3 Virtual Cut-through routing ............................ 14

3 NETSIM Overview 15

3.1 NETSIM Operations ....................................... 16
3.1.1 Module Create Operations .............................. 16
3.1.2 Network Construction Operations ..................... 20
3.1.3 Packet Operations ..................................... 21
3.2 Building switches using NETSIM primitives .............. 23
3.3 Standard Network Generators ........................................... 26
  3.3.1 Delta Network ......................................................... 26
  3.3.2 Binary n-Cube Network .............................................. 27
  3.3.3 n-Dimensional Mesh Network ....................................... 28
3.4 NETSIM Modes ............................................................... 29
  3.4.1 Stand Alone Mode ..................................................... 29
  3.4.2 PARCSIM ............................................................... 29

4 Implementation ............................................................... 31
  4.1 Simulating Wormhole Routing ......................................... 31
    4.1.1 Overview of the Internal Data Structures ..................... 36
    4.1.2 Dynamic process implementation .............................. 39
    4.1.3 Static process implementation ................................. 46
  4.2 Simulating Store and Forward Routing ............................... 50
    4.2.1 Dynamic process implementation .............................. 51
    4.2.2 Static process implementation ................................. 54

5 Performance of the Simulator ............................................. 55
  5.1 Performance of the wormhole routing simulator .................... 56
    5.1.1 Comparison of the two implementation techniques ............ 56
    5.1.2 Analysis of the results ........................................ 59
  5.2 Performance of the store and forward routing simulator .......... 64
    5.2.1 Comparison of the two implementation techniques ............ 64
    5.2.2 Analysis of the results ........................................ 65
  5.3 Comparison of simulation overheads for wormhole and store and
      forward routing .................................................... 66
6 Approximate Models for Simulating Wormhole Routing 68

6.1 Techniques to reduce simulation overhead 68

6.2 Buffer Approximation 70
   6.2.1 Evaluation of the buffer approximation technique 71

6.3 Tail-flit Approximation 76
   6.3.1 Evaluation of the tail-flit approximation technique 78
   6.3.2 Analysis of the results 84

6.4 Combined Approximate Model 85
   6.4.1 Evaluation of the combined approximate model 86
   6.4.2 Analysis of the results 88

6.5 Extension of the combined model 88
   6.5.1 Evaluation of the extended approximate model 89
   6.5.2 Analysis of the results 92

6.6 Comparison of the approximate models 92
   6.6.1 Approximate models for delta networks with small buffers 93
   6.6.2 Approximate models for delta networks with large buffers 93
   6.6.3 Approximate models for mesh networks with small buffers 95
   6.6.4 Approximate models for mesh network with large buffers 95
   6.6.5 Approximate models for cube networks with small buffers 97
   6.6.6 Approximate models for cube networks with large buffers 98
   6.6.7 Overview of the results 99

7 Simulation Model for the iPSC/860 Network 101

7.1 iPSC/860 Network Architecture 101
   7.1.1 Router Architecture 102
   7.1.2 Channel Architecture 103
7.2 Message Passing Protocols ........................................... 103
  7.2.1 Routing Algorithm .............................................. 104
  7.2.2 Routing Operation .............................................. 104
  7.2.3 Message protocol for short messages ....................... 107
  7.2.4 Message protocol for long messages ....................... 107
7.3 NETSIM model for the iPSC/860 network ......................... 108
  7.3.1 Model for the switch and the channel ...................... 108
  7.3.2 Simulation algorithm ........................................... 109
  7.3.3 Simulation model for short messages ...................... 110
  7.3.4 Simulation model for long messages ...................... 110
  7.3.5 Network delays for the simulation model .................. 111
7.4 Results ...................................................................... 116
  7.4.1 Traffic pattern with no contention ....................... 117
  7.4.2 Traffic pattern with contention for one link ............. 118
  7.4.3 Bit-complement traffic pattern .............................. 120
  7.4.4 Matrix transpose traffic pattern ............................ 123
  7.4.5 Summary of the results ........................................ 124

8 Conclusion .................................................................. 126
  8.1 Summary ............................................................... 126
  8.2 Future Work .......................................................... 128

Bibliography .................................................................. 129
Illustrations

2.1 8-Node hypercube network ........................................... 7
2.2 9-Node mesh network .................................................. 8
2.3 8-Node delta network .................................................. 9
2.4 Comparing store and forward and wormhole routing ............ 13

3.1 Basic NETSIM symbols .................................................. 16
3.2 2x2 crossbar switches .................................................. 24
3.3 3-Node ring network .................................................. 25
3.4 4-Node delta network .................................................. 25

4.1 Wormhole routing ..................................................... 32
4.2 Need for delta offset between head and tail flit processes .... 34
4.3 Process associated with the head flit ............................... 41
4.4 Head flit movement from one buffer to the next ................. 42
4.5 Head flit in a blocked state .......................................... 43
4.6 Process associated with the tail flit ............................... 45
4.7 TailEvent moving flits out of a buffer ............................. 46
4.8 Process associated with an input port ............................. 47
4.9 Process associated with a buffer ................................... 48
4.10 Process associated with an output port ........................... 50
4.11 Regular store and forward routing ................................. 51
4.12 Process associated with a packet .................................. 52
5.1 Execution time ratio of static to dynamic implementations for a
256-node delta network ........................................... 57
5.2 Execution time ratio of static to dynamic implementations for a
256-node mesh network .............................................. 58
5.3 Simulator performance as a function of the input load for a 256-node
delta network ....................................................... 59
5.4 Simulator performance as a function of the input load for a 256-node
mesh network ......................................................... 60
5.5 Simulation overheads for a network with single-flit buffers ........... 60
5.6 Difference between the number of processes scheduled by the two
techniques ............................................................ 61
5.7 Process scheduling for single-flit buffers .......................... 62
5.8 Simulation overheads for a network with large buffers ............... 62
5.9 Process scheduling for large buffers ................................ 63
5.10 Execution time ratio of static to dynamic implementations for
256-node networks ................................................... 64
5.11 Simulator performance in cycles per second for 256-node network
models ................................................................. 65
5.12 Comparison of execution times for simulating a 256-node delta network 66
5.13 Comparison of simulation rates for simulating 256-node networks ... 67

6.1 Comparison of latency estimates for a 256-node delta network ....... 69
6.2 Comparison of buffer approximation to the detailed model ............ 70
6.3 Speedup for 256-node networks with a buffer size of four flits ........ 72
6.4 Speedup for 64-node networks with large buffers ..................... 72
6.5 Error in latency for 64-node networks with large buffers .............. 73
6.6 Speedup for 256-node networks with large buffers .......................... 73
6.7 Simulation overheads for a 256-node delta network with large buffers 74
6.8 Error in latency for 256-node networks with large buffers .............. 75
6.9 Performance of buffer approximation as a function of the input load . 76
6.10 Tail-flit approximation with global routing control ..................... 77
6.11 Tail-flit approximation in a buffer ............................................. 78
6.12 Speedup for 64-node networks with single-flit buffers ................ 79
6.13 Error in latency for 64-node networks with single-flit buffers ...... 80
6.14 Speedup for 64-node networks with large buffers ....................... 80
6.15 Error in latency for 64-node networks with large buffers ............ 81
6.16 Speedup for 256-node networks with single-flit buffers .............. 81
6.17 Error in latency for 256-node networks with single-flit buffers ...... 82
6.18 Speedup for 256-node networks with large buffers ..................... 82
6.19 Error in latency for 256-node networks with large buffers ............ 83
6.20 Location of the tail flit when the head flit leaves the network for
   networks with large buffers ...................................................... 83
6.21 Performance of tail-flit approximation as a function of the input load 84
6.22 Simulation overheads for a 256-node delta network with single-flit
   buffers ................................................................................. 84
6.23 Simulation overheads for a 256-node delta network with large buffers 85
6.24 Speedup for 256-node networks with large buffers ..................... 86
6.25 Error in latency for 256-node networks with large buffers ............ 87
6.26 Performance of the combined approximate model as a function of the
   input load ............................................................................. 88
6.27 Simulation Overheads for a 256-node delta network with large buffers 88
6.28 Speedup for 256-node networks with single-flit buffers .............. 89
6.29 Error in latency for 256-node networks with single-flit buffers ....... 90
6.30 Speedup for 256-node networks with large buffers .......................... 90
6.31 Error in latency for 256-node networks with large buffers ............... 91
6.32 Performance of the extended approximation as a function of the input 
load ........................................................................................................ 91
6.33 Simulation overheads for a 256-node delta network with single-flit 
buffers .................................................................................................... 92
6.34 Simulation overheads for a 256-node delta network with large buffers .... 92
6.35 Speedup for 256-node delta network with small buffers ................. 93
6.36 Error in latency for 256-node delta network with single-flit buffers ... 94
6.37 Speedup for 256-node delta network with large buffers ................. 94
6.38 Error in latency for 256-node delta network with large buffers ......... 95
6.39 Speedup for 256-node mesh network with single-flit buffers ........... 96
6.40 Error in latency for 256-node mesh network with single-flit buffers .... 96
6.41 Speedup for 256-node mesh network with large buffers ................. 97
6.42 Error in latency for 256-node mesh network with large buffers ......... 97
6.43 Speedup for 256-node cube network with single-flit buffers ........... 98
6.44 Error in latency for 256-node cube network with single-flit buffers .... 98
6.45 Speedup for 256-node cube network with large buffers ................. 99
6.46 Error in latency for 256-node cube network with large buffers ........ 99
6.47 Best approximate model for networks with large buffers ............... 100

7.1 3-dimensional hypercube network ............................................. 102
7.2 Channel configuration ............................................................... 103
7.3 e-cube routing algorithm ........................................................... 104
7.4 Acknowledgement phase ........................................................... 106
7.5 NETSIM switch model ........................................ 109
7.6 Transmission time for messages less than 100 bytes .......... 112
7.7 Transmission time for messages longer than 100 bytes .......... 114
7.8 Software overheads for various message sizes .................. 116
7.9 Error in the latency prediction for short messages .......... 117
7.10 Error in the latency prediction for long messages .......... 118
7.11 Traffic pattern with contention for one link .................. 119
7.12 Error in the latency prediction for short messages with one-link
    contention traffic pattern ...................................... 119
7.13 Error in the latency prediction for long messages with one-link
    contention traffic pattern ...................................... 120
7.14 Bit-complemented destination addresses for an 8-node hypercube . 121
7.15 Error in the latency prediction for saturated bit-complement traffic
    for short messages ............................................ 121
7.16 Error in the latency prediction for saturated bit-complement traffic
    for long messages ............................................ 122
7.17 Error in the latency prediction for lightly loaded bit-complement traffic
    for short messages ............................................ 122
7.18 Error in the latency prediction for lightly loaded bit-complement traffic
    for long messages ............................................ 123
7.19 Matrix transpose traffic pattern for a 16-node hypercube ........ 124
7.20 Error in the latency prediction for lightly loaded matrix transpose
    traffic for short messages .................................... 125
7.21 Error in the latency prediction for lightly loaded matrix transpose
    traffic for long messages .................................... 125
Chapter 1

Introduction

An interconnection network is a key component in the design of a parallel processing system. It must provide fast and efficient communication between the processing nodes. The performance of the interconnection network directly impacts the efficiency of the entire system. An interconnection network can slow down the entire system, causing processing nodes to remain idle, inefficiently utilizing system resources. The performance of an interconnection network depends on several factors, such as the topology, the switching strategy, the routing and arbitration protocols, the switch architecture used in the network and the input load applied to the network. Extensive research (described below) has been undertaken to study the interaction of these factors in the design of an efficient interconnection network. This thesis presents a modular, general purpose network simulation tool that can be used in this study.

1.1 Motivation and Objective

To evaluate the interaction of the design factors, several approaches to performance prediction, which range from analytical techniques to experiments with prototypes, have been developed. Analytical techniques use the fastest and the cheapest approaches to modeling an interconnection network. Several analytical models have been developed to study the various factors that effect the performance of interconnection networks [2], [9], [13], [21], [20]. One of the disadvantages of this technique is the approximation of the load applied to the network. Most of the analytical techniques assume a stochastic load with the traffic uniformly distributed among all the nodes. The network load in real systems running applications is usually not uniform.
The analytical models also make several simplifying assumptions about the switch architecture to keep the model tractable. For example, most of the analytical models use a simple switching scheme like store and forward routing [9], [13], [20]. Models that use complex switching schemes, like the wormhole routing [8], [2] assume a simple traffic distribution pattern and a simple switch architecture. Other models ignore the effect of contention in the network. As a result, analytical techniques are not suitable for predicting the performance of large networks with complex switches and complex switching schemes.

Experimentation with prototypes, the other extreme approach to performance evaluation, permits network performance to be studied in detail. However, the prototype systems are expensive, inflexible and usually small. Because of their limited size, they cannot accurately estimate performance of complex switching schemes, which depends on the amount of contention in the network which in turn depends on the size of the system.

A simulation model is an attractive tool for evaluating the performance of networks. It is extremely flexible and can be used to model complex network architectures and switching schemes to any desired detail, constrained only by the available computing resources. A simulation model can be driven either by stochastic loads or loads generated by the execution of real applications on a parallel system. It can also model network contention in detail.

The objective of the research described in this thesis is to develop an interconnection network simulator that can efficiently simulate various switching strategies including complex ones like wormhole routing in large-scale networks. The simulator also allows the user to specify different network topologies, different routing schemes and different switch architectures.
This thesis presents a network simulator called NETSIM. NETSIM is a modular, general purpose interconnection network simulator developed as an extension to YACSIM [18], a C-language based process-oriented discrete event simulator, to simulate large-scale networks efficiently. NETSIM provides five basic network building modules that can be connected in any desired topology. It also allows the user to simulate different switching schemes like wormhole routing, virtual cut-through routing and store and forward routing. The network models can either be simulated in isolation, or be driven by the execution of application programs in an execution-driven simulation environment. This thesis emphasizes efficient techniques to simulate complex switching schemes such as wormhole routing. It evaluates the relative overheads associated with different implementation schemes. This thesis also develops four approximate models for simulating wormhole routing, and evaluates the trade-off between accuracy and speedup.

Validation of a simulation model is necessary to gain confidence in the accuracy of its predictions. This thesis develops a model for the iPSC/860 [27] hypercube interconnection network using the NETSIM primitives. The message latencies predicted by the NETSIM model for various traffic patterns are compared with the actual message latencies measured on the iPSC/860 hypercube.

1.2 Related Work

Several network simulators have been developed to characterize different network architectures. Chittor et al. [4] have developed a high-performance simulator for large wormhole-routed networks. This simulator is implemented directly in the C language, and models wormhole routing in direct networks. The simulator does not provide any capability to simulate other routing schemes such as virtual cut-through and store
and forward. It also does not allow the users to change the switch architecture. A simulator shell simulates the behavior of the individual tasks of a parallel application running on different nodes. The simulator is not used in an execution-driven simulation environment. MultiSim [24], another network simulator written in C and C++, was developed with the event-driven simulation package CSIM [32]. MultiSim assigns a single CSIM process to each packet to model its movement through the network. MultiSim uses two different algorithms to simulate wormhole routing in networks with single flit buffers and multiple flit buffers, and assumes that the free space availability in the switch buffers is signaled globally. In addition to implementing wormhole routing, MultiSim also implements multicast and virtual channels. Most of the other network simulators discussed in the literature are restricted in their application, having been developed to validate analytical models.

1.3 Overview of the Thesis

Chapter 2 reviews interconnection network topologies and summarizes routing strategies used in interconnection networks. Chapter 3 presents an overview of NETSIM and describes the user interface provided by NETSIM. Chapter 4 gives the implementation details of the simulator, and describes the different implementation techniques. Chapter 5 evaluates the implementation techniques described in Chapter 4. Chapter 6 discusses the approximate models used for simulating networks and evaluates the tradeoff between accuracy and speedup obtained by the approximate models. Chapter 7 presents a simulation model for the iPSC/860 hypercube network and validates the simulator. Chapter 8 summarizes the results of the different implementation techniques and the different approximate techniques, and suggests future extensions to the simulator.
Chapter 2

Interconnection Networks

Parallel processing systems built with hundreds or thousands of general purpose microprocessors have emerged as a promising means to achieve high performance computing. Such systems have been broadly classified into local memory multiprocessor systems and global memory multiprocessor systems [3]. A local memory multiprocessor system [25] consists of processing nodes connected by an interconnection network. An individual processing node in such a system contains a microprocessor, local memory and an interface to the network. Programs executing on a node interact with the other nodes by explicitly sending messages across the interconnection network. A global memory multiprocessor system also consists of a number of processing nodes, but it provides a single global address space for all applications running on the processing nodes. Programs interact implicitly through shared variables. In such a system, the global memory is either distributed among the processing nodes, with the processing nodes being connected by an interconnection network [11], or it is centrally located, with an interconnection network providing the interface between the processing nodes and one or several memory modules. The interconnection network forms a critical component of such a parallel processing system. It must provide fast and efficient communication at a reasonable cost. The performance of the entire system depends heavily on the performance of the interconnection network.

Ideally, it would be desirable to connect every node in the system to every other node. Although a complete crossbar interconnection scheme makes the physical separation between the modules transparent to the application, it is very expensive, requiring \( N^2 \) links to connect \( N \) modules. Such an interconnection scheme is not
feasible for a large number of nodes. The other extreme would be to provide a single path, a bus, that all the modules share. A bus is the least expensive interconnection scheme but it is also the slowest. Only a small number of modules can share the bus efficiently.

Interconnection schemes that lie between these two extremes provide a tradeoff between performance and cost. They provide communication paths between every pair of nodes in the system, but require some paths to share the network resources. Since every node in the system is not directly connected to every other node, the communication paths between nodes often pass through a sequence of intermediate nodes.

An interconnection structure is characterized by its topology, routing scheme, flow control and the switching mechanism. These characteristics are explained in the following sections.

2.1 Topology

An interconnection network consists of switches and links. A switch is also referred to as a router, and has one or more input terminals and one or more output terminals and is capable of selectively transferring data from any input terminal to any output terminal. Links, also called channels, provide the communication paths and are used to connect the output terminals of switches to input terminals of other switches.

The topology defines how the switches are interconnected by the links. The topology is represented by a graph with the switches forming the nodes of the graph and the communication links forming the edges. Several interconnection topologies that provide a wide range of tradeoffs between performance and cost have been proposed and classified [12].
Interconnection networks can also be classified into direct networks and indirect networks [1] based on how the switches and links are connected to the processing elements.

2.1.1 Direct Networks

A direct network has every switch associated with a module in the network. Each switch has at least one terminal and link that connects it to the module, which is referred to as the internal channel, and provides the interface between the module and the network. The other terminals of the switch are directly connected to other switches in the network. The links between switches are referred to as external communication channels. If the switches attached to two modules share a link, the directly connected modules are called neighboring or adjacent modules. Adjacent modules can communicate by passing data through only one link. Non-adjacent modules communicate by establishing a path consisting of two or more links between the two modules. Examples of direct networks are mesh networks, binary n-cube networks and ring networks. A binary 3-cube network used to interconnect the processors of a 8-node hypercube multicomputer [25] is shown in Figure 2.1. A 2-dimensional mesh

![8-Node hypercube network](image)

Figure 2.1: 8-Node hypercube network
interconnection scheme used to connect a 9-node mesh-based multicomputer is shown in Figure 2.2

Figure 2.2: 9-Node mesh network

2.1.2 Indirect Networks

In an indirect network, only some of the switches are connected to modules. Switches not connected to any module are connected to other switches in the network. A multistage interconnection network is an example of an indirect network. Examples of multistage interconnection networks include baseline networks [36], omega networks [22], indirect binary n-cube networks [29] and delta networks [28]. An 8-node delta network is shown in Figure 2.3. Data is transferred into the first stage of the network by the module. Then it proceeds from switches in one stage to the switches in the other stage until it reaches the other side of the network. As shown in Figure 2.3, the data must pass through every stage before reaching the destination.
2.2 Routing Protocol and Flow Control

The topology of an interconnection network should allow every node in the system to communicate with every other node in the system. For two nodes to communicate, a *path* has to be established between them. A *path* is defined as a sequence of communication links from the source node to the destination node. A routing algorithm builds this communication path between the source node and the destination node.

A routing algorithm can be implemented either in software or in hardware. It can be designed such that it uses either local control or requires global state information. Providing global state information to each switch in the network is expensive. In order to minimize the time taken to make a routing decision, an efficient routing algorithm is usually implemented in hardware and uses only local control.

Routing algorithms are classified as deterministic or adaptive. In a deterministic routing scheme, also known as an oblivious routing scheme, a unique path exists between two nodes, and this path is used for all communication between the nodes.
An adaptive routing scheme, or a non-oblivious routing scheme, can use one of several paths between two nodes in the system. A particular path can be chosen to reduce link congestion.

Two or more communication paths through a switch can contend for the same output link at the same time. The routing controller in the switch also arbitrates between conflicting requests and awards the network link to one of the requests.

If the switch has no space allocated to hold blocked data, the partial path from the source node up to that switch is torn down and the source must retry. However, if the switch contains buffers, the blocked data remains in the switch waiting for the link. The flow control protocol determines how the buffers and links are allocated to data as it travels along a path in the network. It also determines how a data collision is handled. It regulates the transmission of data to prevent buffer overflow. If all the buffers along a path fill up, the flow control protocol provides a means to throttle the data traffic. The buffers can be placed in various positions inside the switch. They can be implemented as simple FIFO queues or complex split queues [35].

Instead of transmitting a single stream of data along a physical communication link, several data streams can be multiplexed on to the same physical link. Each such data stream is referred to as a virtual channel [6]. In a network using virtual channels, the buffers associated with each physical channel are organized into several lanes, each lane corresponding to a virtual channel.

The virtual channel flow control makes efficient use of the physical bandwidth of the communication links by allowing data in one lane to pass blocked data in another lane.
2.3 Switching Strategies

Switching is the mechanism by which data is actually transferred from an input terminal to an output terminal inside the switch, after the routing algorithm specifies the output channel and the arbitration algorithm resolves the conflicts. To use the network resources efficiently, a switching protocol usually organizes the data into packets. A packet is the smallest unit which carries routing and sequencing information. A packet is further organized into several smaller units of data called flow control digits, or flits [7]. A flit is the smallest unit of data that can be transferred between two switches in a single cycle. The size of a flit is determined by the width of the link connecting the two switches. The first flit of a packet, referred to as the header or head flit, contains the routing information used by the routing algorithm implemented in the switch controller to select one of the output terminals. The remaining flits of the packet carry no routing information and follow the path established by the head flit. The last flit of the packet is the tail flit, and it releases the channels acquired by the packet, by freeing each switch output terminal it traverses.

When a packet moves from a source node to a destination node, the flits of the packet can be spread over several switches in the network. The number of switches over which a packet can be spread is precisely determined by the switching protocol. Store and forward switching allows a packet to be spread over at most two switches. The wormhole routing and the virtual cut-through routing protocols allow the flits to be spread over any number of switches. Therefore, in a network using these protocols, a packet could be spread over all the switches that make up the path from the source node to the destination node.

The switching strategy has a direct impact on an important metric used to evaluate the efficiency of a network, the communication latency. Latency is the time taken for
a packet to move from the source node to the destination node. It includes the time involved in initiating the packet transfer, referred to as the start-up time, the time actually spent in the network establishing the path and transferring the data, and the delay incurred due to contention for channels.

2.3.1 Store and forward routing

The first generation multicomputers based on the Cosmic Cube [33] (e.g., the iPSC-1 [31] and the nCUBE-1 [14]) used a switching strategy called store and forward switching. This technique moves all the flits of a packet into a buffer in a switch, before any of the flits are forwarded to the next switch. After the whole packet is stored in a buffer, it is forwarded only when the next channel is available and the next switch has enough buffer space to store the entire packet. Using this technique, the flits of a packet can be spread over at most two switches in the network. The store and forward technique is simple to implement. One limitation of the store and forward technique is its buffer space requirement. Each switch needs to buffer every incoming packet. This also puts a limit on the maximum packet size that the network can handle. A more serious drawback is the increased network latency for packets going to non-adjacent nodes. Since the entire packet is buffered at every switch before it is forwarded, the network latency is proportional to the distance between the source and the destination. If L is the length of the packet, and D is the distance between the source and the destination, store and forward give a latency which depends on the product of L and D.

2.3.2 Wormhole routing

This technique was proposed by Dally and Seitz [7] and is implemented in most of the second generation multicomputers like the nCUBE-2 [25] and the Touchstone
DELTA [16]. This technique is characterized by the property that the head and the following flits of a packet can leave a switch as soon the packet acquires the output channel and the next switch has buffer space. The head flit need not wait for the tail flit to catch up, before moving forward. The wormhole technique requires that the switches have a buffer to hold at least one flit. The head flit can advance even if the next buffer has only one free flit position. When the header flit encounters a busy channel, the entire packet is blocked in place. The flits remain in the buffers that the packet has already acquired. The main advantage of the wormhole technique over store and forward is the reduced latency for non-adjacent nodes. The latency is reduced because the transmission of the flits is pipelined. Given that the length of the packet is $L$ and it traverses $D$ channels between the source and the destination, the latency is proportional to the sum of $L$ and $D$. Figure 2.4 compares the network latency incurred for the wormhole and store and forward techniques. The wormhole

![Figure 2.4: Comparing store and forward and wormhole routing](image)

technique also reduces the buffer space requirements. Wormhole switches require
only a few flit buffers per switch, making it possible to design fast and inexpensive switches.

2.3.3 Virtual Cut-through routing

The virtual cut-through technique [19] is similar to the wormhole technique discussed in Section 2.3.2. The head flit of a packet does not wait for the tail flit to catch up. However, in virtual cut-through the head flit is forwarded from one switch to the next only if the destination switch has enough buffer space to store the entire packet. The flits of a blocked packet eventually collect in one switch and the packet is removed from the network. The network latency using this technique is proportional to the sum of L and D.
Chapter 3

NETSIM Overview

NETSIM is developed using the primitives provided by YACSIM [18], a process-oriented, discrete event simulator based on the C programming language. NETSIM provides data structures and subroutines that can be used to construct and simulate a wide range of interconnection network topologies.

NETSIM is modular and provides the five basic network building blocks shown in Figure 3.1. These basic modules can be interconnected in different ways to construct different types of switches.

- **Multiplexer**: This is a multi-input, single-output module used to merge data. This module resolves conflicts among several of its input terminals trying to transfer data simultaneously to its output terminal. The arbitration algorithm is currently implemented as a semaphore with a FIFO queuing discipline.

- **Demultiplexer**: This is a single-input, multi-output module used to route data along one of several possible paths. This module implements the routing mechanism of the network. Every time a demultiplexer module is created, a routing function is assigned to it. The demultiplexer uses information in the header of the packet at its input terminal and the routing function to determine the output terminal through which the packet will leave the module.

- **Buffer**: This module is used to provide temporary storage for message flits as they move through the network. The buffer is implemented as a finite FIFO queue. When a buffer is created, it is assigned an integer value which specifies the number of flits that it can buffer. The buffer is also assigned a threshold
value, which specifies the number of free flit positions required before another flit can move into the buffer.

- **Network Ports:** These modules provide the interface between the processing elements and the network. The input port controls the movement of packets from the processing node to the network. The output port removes packets from the network.

3.1 **NETSIM Operations**

NETSIM provides subroutines to create and manage the basic network modules.

3.1.1 **Module Create Operations**

This section describes the operations that create new NETSIM modules.
• **BUFFER**  \( *\text{NewBuffer}(id,\text{size},\text{threshold}) \)

\[ \begin{align*}
  &\text{int } id; \\
  &\text{int } size; \\
  &\text{int } threshold;
\end{align*} \]

This operation creates and returns a pointer to a new network buffer. It assigns the buffer the number \( id \) which is used to identify the buffer in trace statements. \( \text{Size} \) specifies the maximum number of flits that can be stored in the buffer, and \( \text{threshold} \) is the number of flit positions that must be empty before a new flit can enter the buffer.

• **MUX**  \( *\text{NewMux}(id,\text{fanin}) \)

\[ \begin{align*}
  &\text{int } id; \\
  &\text{int } fanin;
\end{align*} \]

This operation creates and returns a pointer to a new multiplexer with \( \text{fanin} \) input terminals and the module identifier \( id \).

• **DEMUX**  \( *\text{NewDemux}(id,\text{fan.out},\text{routingfcn}) \)

\[ \begin{align*}
  &\text{int } id; \\
  &\text{int } fan\text{.out}; \\
  &\text{rtfunc } routingfcn;
\end{align*} \]

This operation creates and returns a pointer to a new demultiplexer with \( \text{fan.out} \) output terminals and identifier \( id \). It also assigns the function \( \text{routingfcn} \) to be used by the demultiplexer in routing the packet at its input terminal to one of its output terminals. The type \( \text{rtfunc} \) is a pointer to a function with three arguments that returns an integer. It is called with the following arguments each time a packet enters a demultiplexer:
src - a pointer to the id of the source processor sending the packet.

dest - a pointer to the id of the destination processor receiving the packet.

demuxid - the id of the demultiplexer.

The user must provide a function that can compute the demultiplexer output terminal from these parameters. Pointers to the ids of the source and destination modules are passed so that the routing function can change the source and destination fields.

- **IPORT**  
  \*NewIPort(id, size)

  int id;

  int size;

  This operation creates and returns a pointer to a new network input port with identifier *id* and the capacity to buffer up to *size* packets. The size of the port is specified in terms of the total number of packets it can hold.

- **OPORT**  
  \*NewOPort(id, size)

  int id;

  int size;

  This operation creates and returns a pointer to a new network output port with identifier *id* and the capacity to buffer up to *size* packets.

- **SEMAPHORE**  
  \*IPortSemaphore(iptr)

  IPORT *iptr;

  This operation returns a pointer to a semaphore within the port pointed to by *iptr*. If the port is full and cannot accept new packets, a sending activity can wait on this
semaphore. When space becomes available, the port will signal the semaphore and release the activity.

- \textit{int}\quad \textit{IPortSpace}(\textit{port})

\textit{IPORT*port;}

This operation returns the number of packets that can be sent to the input port pointed to by \textit{port} before its queue becomes full.

- \textit{int}\quad \textit{IPortGetId}(\textit{port})

\textit{IPORT*port;}

This operation returns the \textit{id} of the port pointed to by \textit{port}.

- \textbf{SEMAPHORE}\quad \textit{*OPortSemaphore}(\textit{optr})

\textit{OPORT*optr;}

This operation returns a pointer to a semaphore within the port pointed to by \textit{optr}. If the port is empty, an activity can wait on this semaphore. When a packet becomes available, the port will signal the semaphore and release the activity.

- \textit{int}\quad \textit{OPortPackets}(\textit{port})

\textit{OPORT*port;}

This operation returns the number of packets that are currently queued at the network output port pointed to by \textit{port}.

- \textit{int}\quad \textit{OPortGetId}(\textit{port})

\textit{OPORT*port;}

This operation returns the \textit{id} of the port pointed to by \textit{port}. 
3.1.2 Network Construction Operations

The following routines provide the capability to build different network modules and assign various parameters to model different delays associated with the network.

- **void** `NetworkConnect(src, dest, src_index, dest_index)`
  
  `MODULE *src;`
  
  `MODULE *dest;`
  
  `int src_index;`
  
  `int dest_index;`

  This operation is used to build a network by interconnecting the basic modules (buffers, multiplexers, demultiplexers, and ports). It connects an output terminal of the module pointed to by `src` to an input terminal of the module pointed to by `dest`. If the source module is a demultiplexer, then `src_index` specifies which output terminal is used. This parameter is ignored for all other modules since they have only one output terminal. The parameter `dest_index` designates an input terminal for the destination module if it is a multiplexer, and is ignored for all other types of modules.

- **void** `NetworkSetFlitDelay(t)`
  
  `double t;`

  Execution of this operation sets the time required to move a flit from a buffer or a port to another buffer or port to `t`. This value is used for all buffers in the network.

- **void** `NetworkSetMuxDelay(i)`
  
  `int i;`

  Execution of this operation sets the time required to move a flit through a multiplexer to `i` times the flit delay.
• void NetworkSetArbDelay(i)

int i;
Execution of this operation sets the time for a multiplexer to arbitrate between competing flits to \( i \) times the flit delay.

• void NetworkSetDemuxDelay(i)

int i;
Execution of this operation sets the time required to move a flit through a demultiplexer to \( i \) times the flit delay.

3.1.3 Packet Operations

In addition to providing routines to create and manage the network modules, NETSIM also provides routines to create and manage packets within the network

• PACKET NewPacket(seqno,msgptr,sz,src,dest)

int seqno;
MESSAGE *msgptr;
int sz;
int src;
int dest;

This operation creates and returns a pointer to a new packet and assigns it the sequence number \( \text{seqno} \). \text{Msgptr} \) is a pointer to the message that the packet is a part of, and \( \text{sz} \) is the number of flits in the packet. The identifier of the processor sending the packet is given by \( \text{src} \) and \( \text{dest} \) is the identifier of the receiving processor.

• int PacketSend(pkt,port)

PACKET *pkt;
IPORT *port;

Execution of this operation tries to send the packet pointed to by pkt into a network port pointed to by port. If the operation is successful, PacketSend returns a value of 1. However, if the port queue is full, this operation returns a value of 0.

- PACKET PacketReceive(port)

OPORT *port;

This operation attempts to remove a packet from a network through the port pointed to by port. If there is no packet available at this port, it returns the NULL pointer.

- void PacketStatus(pkt)

PACKET *pkt;

This operation prints the status of the packet pointed to by pkt. It shows the total number of free flit positions (bubbles) and the number of flits in each buffer between the head and the tail flits of the packet.

- PKTDATA PacketGetData(pkt)

PACKET *pkt;

This operation returns a pointer to the user accessible data of the packet data structure pointed to by pkt. The data structure returned to the user contains the following fields.

- an integer specifying the sequence number of the packet within a message.
- the ID of the node sending the packet.
- the ID of the node receiving the packet.
- the size of the packet in flits
• a time stamp, specifying the time the packet entered the network.

• a pointer to the message that the packet belongs to.

3.2 Building switches using NETSIM primitives

The NETSIM modules are used to build different types of switches. Switches are commonly implemented as a full crossbar switch to allow all possible connections between its inputs and outputs. Switches can be unbuffered or can have buffers at the input, output or inside the crossbar switch. Figure 3.2 shows different types of 2x2 crossbar switches built using the NETSIM modules with different buffer configurations. A routine to generate a 2x2 unbuffered switch is given below.

```c
MUX *mux1,*mux2;
DEMUX *demux1,*demux2;
mux1 = NewMux(0,2);
mux2 = NewMux(1,2);
demux1 = NewDemux(0,2, routingfcn);
demux2 = NewDemux(1,2, routingfcn);
NetworkConnect(demux1, mux1, 0, 0);
NetworkConnect(demux1, mux2, 1, 0);
NetworkConnect(demux2, mux1, 0, 1);
NetworkConnect(demux2, mux2, 1, 1);
```

The module create operations are used to create new multiplexer and demultiplexer modules. The network construction operation is used to wire the modules.
The unbuffered switch can be easily extended by adding buffers at the output as shown below.

\[
\text{BUFFER } *\text{buf1}, *\text{buf2};
\]
\[
\text{buf1} = \text{NewBuffer}(0, 5, 1);
\]
\[
\text{buf2} = \text{NewBuffer}(1, 5, 1);
\]
\[
\text{NetworkConnect}(\text{mux1, buf1, 0, 0});
\]
\[
\text{NetworkConnect}(\text{mux2, buf2, 0, 0});
\]

These routers can be used as building blocks to wire up any arbitrary network. Figure 3.3 shows a 3-node ring network with unidirectional channels built using the unbuffered switch shown in Figure 3.2. Figure 3.4 shows a 4-node delta network built using the output buffered switch shown in Figure 3.2.
Figure 3.3: 3-Node ring network

Figure 3.4: 4-Node delta network
3.3 Standard Network Generators

In addition to providing the capability to built any arbitrary network, NETSIM also provides routines to generate some standard networks like binary n-cube networks, n-dimensional mesh networks and n-stage delta networks.

3.3.1 Delta Network

This routine generates an n-stage delta network built from crossbars switches with a inputs and b outputs. Such a network connects a^n source modules to b^n destination modules. A special type of delta network consists of square crossbar switches with a equal to b. The following NETSIM routine builds a delta network .

- void NewDeltaNet(sufunc, radiz, stages, netin, netout, bufsize, threshold, portsize)

switchfunction sufunc;
int radiz;
int stages;
IPORT **netin;
OPORT **netout;
int bufsize;
int threshold;
int portsize;

The type switchfunction is a pointer to a function that returns a pointer to a switch module. A switch module contains multiplexers, demultiplexers and buffers connected in the manner specified by the function sufunc. Different functions can be used to return different switch configurations as shown in the two examples in Section 3.2. The size of each switch in the network is specified by radiz. The switch
function is called with the following input parameters, the size of the switch, the size of the buffers in the switch specified by bufsize and the threshold value for the buffers specified by threshold. netin and netout are two arrays that contain pointers to the input ports and the output ports. The routine takes in two empty arrays as input parameters. It creates an input port and an output port associated with each node of the network. The size of the ports is specified by portsize. It fills the two arrays with the pointers to the ports. The number of stages in the network is specified by stages.

3.3.2 Binary n-Cube Network

This routine generates a n-dimensional hypercube.

   • void NewHypercubeNet(sufunc, dim, netin, netout, bufsize, threshold, portsize)
   
   switchfunction sufucn;
   int dim;
   IPORT **netin;
   OPORT **netout;
   int bufsize;
   int threshold;
   int portsize;

   The dimension of the binary n-cube is specified by dim. The type switchfunction is a pointer to a function that takes the dimension, the buffer size specified by bufsize and the buffer threshold specified by threshold as input parameters and returns a pointer to a switch module. The parameters netin and netout are pointers to two empty arrays that are pointers to port modules. The routine creates ports with size
specified by \textit{portsize} and returns pointers to the input ports and output ports by filling up the arrays \textit{netin} and \textit{netout}.

\subsection*{3.3.3 \ n-Dimensional Mesh Network}

This routine generates a \textit{n}-dimensional mesh.


\begin{verbatim}
swfunc
int dim;
int *size;
IMPORT **netin;
OPORT **netout;
int bufsize;
int threshold;
int portsize;
\end{verbatim}

The dimension of the mesh is specified by \textit{dim}. \textit{Size} is a pointer to an array that contains the number of nodes in each dimension. \textit{Switchfunction} is a pointer to a function that takes the dimension of the mesh, the size of the mesh, the size of the buffer specified by \textit{bufsize} and the buffer threshold specified by \textit{threshold} and returns a pointer to a switch module. The routine takes in two empty arrays that are pointers to port modules. It creates ports with size specified by \textit{portsize} and returns pointers to the input ports and output ports by filling up the arrays \textit{netin} and \textit{netout}. 
3.4 NETSIM Modes

The NETSIM network models can be used in two different operating environments. The two modes differ in the manner in which the packets are generated.

3.4.1 Stand Alone Mode

A stand-alone model of a network is used to characterize the network parameters such as latency, queuing delays, channel utilization and throughput by driving it with stochastic loads. The load for the model is generated by YACSIM processes. A YACSIM process is created for every port in the network model. This process is scheduled to wait for the port. When an input port is ready to accept a packet, the process creates a new packet and sends it out through the port. The interpacket delay time is selected based on any required distribution. This time depends on the required input load characteristics. The process delays for this time before trying to put another packet into the network. The process can use any specified distribution function to select the packet destinations.

The network signals an output port when a packet is available at that port. YACSIM processes associated with the output ports remove the packet from the port and collect all the desired statistics.

3.4.2 PARCSIM

NETSIM can also be used as an integral part of PARCSIM, a parallel architecture simulator. PARCSIM is also an extension to YACSIM [18] and provides the capability to create processors and attach processes, memory modules, caches and ports to these processors. The network model is used to connect the processors in any desired topology. The network model is interfaced to the processors by the ports attached
to the processors. The entire architecture model is then used to simulate a complete parallel system using the execution-driven simulation technique. In such a system, the simulation is driven by the execution of real parallel programs. These parallel programs are instrumented by profilers to extract timing information and the profiled code is then simulated by PARCSIM. Each target architecture has a profiler and the profilers along with PARCSIM constitutes the Rice Parallel Processing Testbed, RPPT [5]. Therefore, in an execution-driven simulation environment, the network traffic is generated by the execution of real programs.
Chapter 4

Implementation

An interconnection network simulator has to be efficient both in terms of time and space in order to simulate large networks with a reasonable overhead. The approach of scanning a set of switches and moving all the packets that are ready to move can be inefficient, especially when simulating large, heavily-loaded networks. NETSIM uses several different approaches to allocate and manage the simulation processes. The techniques vary, depending on the kind of routing strategy used in the network. This chapter describes the implementation details of simulating networks that use wormhole routing and store and forward routing. Techniques used for simulating networks that employ virtual cut-through routing are similar to the techniques used in simulating wormhole routed networks. All the techniques discussed here are independent of the length of the packet and the buffer size.

4.1 Simulating Wormhole Routing

Wormhole routing is characterized by the property that the flits of a packet can get spread over several switches in the network, as discussed in Section 2.3.2. The individual flits of a packet can move asynchronously depending on the availability of buffer space between the head and the tail flits. This introduces additional complexity in simulating wormhole routing. Figure 4.1 shows a wormhole network with four switching stages, each stage having a buffer capable of holding one flit. The packet consists of five flits. Assume the head flit gets blocked at time t=2. The intermediate flits continue to move until they fill up the available buffer space between the head and the tail flits. Free flit positions between the head and the tail flits are referred to
as bubbles in the rest of this thesis. Now assume that the head flit acquires the next channel and moves to the next switch at time $t=8$. Each of the remaining flits starts to move only when there is a bubble in front of it. In the usual implementations of wormhole routing, there is no global controller which keeps track of the number of free flit positions in the individual buffers. The information about the availability of free space has to be explicitly transmitted between two stages using hand shake signals. This precludes the possibility of a flit moving into a full buffer at the same time that another flit is moving out, and is referred to as the local control protocol for the transfer of flits.
Since every flit in the packet can move independently, this movement must be modeled to develop an accurate simulator. Assigning a single YACSIM simulation object for every flit in the packet would be a brute-force approach to simulating such a system. Assume a 256-node network with each node injecting a 100-flit packet into the network. The brute force approach would require 25600 simulation objects to represent the system, and the computing power required to manage the system would be enormous. Such an approach would severely limit the size of the network that can be simulated efficiently with the available computing resources. Therefore, two new simulation organizations have been developed to implement wormhole routing in NETSIM. The first technique is called the **dynamic process implementation**. This technique assigns two simulation processes for each packet, one associated with the head flit of the packet and the other associated with the tail flit of the packet. The second technique is called the **static process implementation**, and it assigns a single simulation process to each port and each buffer in the network. These processes model the movement of the head and the tail flits of the packet.

Another problem exists in this approach of modeling the entire packet based on the head and tail flits. This problem is inherent in both implementations. The processes associated with moving the head and tail flits of a packet can be active at the same simulation time, and can modify the packet, buffer and port data structures. As a result, the availability of free space in buffers can be signaled at the wrong time, depending on the order in which the processes come off of the event list. This problem is illustrated in Figure 4.2.

The network has packets consisting of six flits each, with buffers capable of storing five flits. The flit distribution at time $t = 1.0$ is shown in the figure. If the processes associated with moving the head and tail flits are scheduled at the same simulation time, the following sequence of events can occur. The process associated with the
Figure 4.2: Need for delta offset between head and tail flit processes

head flit in buffer 3 schedules a move into the next buffer, buffer 4. The tail flit of that packet, in buffer 2, also schedules a move of one of the intermediate flits, into buffer 3. If the process associated with the head flit in buffer 1 comes off of the event list behind the tail flit in buffer 2, it sees the free space created and schedules a move for the next cycle. The result is that a flit moves into buffer 2 and at the same time a flit moves out from buffer 2. If this head had been scheduled before the tail flit, it would have encountered a full buffer and suspended. In order to eliminate this race condition, the process associated with moving the head flit of a packet is scheduled at simulation times that are offset from the times the process associated with moving the rest of the flits of the packet, by a small delay, delta. This eliminates the possibility of the two processes being scheduled at the same time and modifying the same data structures.

In the example in Figure 4.2, the processes associated with the head flits are scheduled at integral multiples of simulation time. The processes associated with the
tail flits are staggered by an offset value of 0.1. The head flit in buffer 3 schedules a move at time $t=1.0$ which completes at time $t=2.0$. The head flit in buffer 1 is also active at this simulation time but encounters a full buffer, buffer 2, and suspends. The tail flit in buffer 2 schedules a move at time $t = 1.1$ and creates the free space in that buffer which is seen by the head flit in buffer 1 at the correct simulation time, $t= 2.0$. The head flit in buffer 2 then schedules a move to start at time $t=2.0$ cycles.

Using only two processes per packet instead of using one process per flit can still require a large number of processes. Since each simulation process must maintain its own private stack, a significant amount of memory may be used in a simulation. The amount of stack required depends on the compiler. On an IBM RS/6000 machine, each YACSIM processes requires a 5000-byte stack, while on a SPARC machine, each YACSIM process requires about 20,000 to 30,000-byte stack. This limits the size of the network that can be simulated using the available computing resources. Therefore, in order to use the approaches detailed above, it is necessary to further reduce the memory requirements. YACSIM provides an effective mechanism to achieve this objective.

YACSIM provides two types of objects, processes and events, to represent the active elements of a system. Each is defined by a procedure, called the body. This procedure is executed when the process or event is activated. However, a process can suspend at any point in its body code, and then resume execution at the same point at a later time. The process saves all its local variables on its own stack when it suspends, so they are available when the process resumes. When an event is activated, its body procedure is executed, starting at the beginning. It can not suspend since it does not have a separate stack for its local variables. When the event occurs again, it must start over at the beginning of its body procedure. It can not even use static variable, since there may be several events that use the same body procedure.
In order to use events to mimic processes, the events must be provided with the ability to resume execution at different points in the body code. This is done by using a state variable that is saved in the event descriptor each time the event's body procedure returns. Different values are saved for the different points in its body procedure at which the event can return. When the event resumes execution, a case statement at the start of its code uses the state variable information to branch to the point of previous suspension. The problem with local variables is solved by passing a pointer to a data structure as an argument to the event. All the local variables are made part of the data structure.

This is a very limited emulation of a process. For example, an event's body procedure can not return from within a nested subroutine. However, it offers significant savings in memory space over the use of processes. For this reason, all of the processes mentioned in the two implementations discussed in this chapter are realized as events. The terms process and event are used interchangeably to refer to a YACSIM event in the implementation.

4.1.1 Overview of the Internal Data Structures

The NETSIM routines modify the data structures associated with the packet and the buffer and port modules to model the flow control within the network. This section lists these data structures.

Packet data structure

The user accessible part of the packet data structure is defined in Section 3.1.3. In addition, the following fields are used to model the movement of the flits.
• an integer field to keep track of the amount of free space between the head and the tail flits.

• an integer field to represent the offset of the tail flit from the front of the buffer. This determines the number of intermediate flits that a process has to move before moving the tail flit from one module to another.

• a pointer to the event associated with the head flit of the packet in the dynamic implementation.

• a pointer to the event associated with the tail flit of the packet in the dynamic implementation if the event is waiting for the head flit to move. This pointer is referred to as the pointer to a waitingtail, and it is set to NULL if the process associated with the tail flit of the packet is active. It is set to point to that process when the number of bubbles between the head and the tail flits is reduced to zero, and the tail flit cannot advance any further unless the head flit moves.

• pointers to modules that contain the head and the tail flits of the packet, which can either be a port or a buffer. These modules are referred to as the headbuffer and the tailbuffer respectively, in the subsequent discussion.

• a pointer to the input port through which the packet enters the network.

Buffer data structure

The user sets the id, the size of the buffer and the threshold value with the NewBuffer call. The following fields are used to manage the buffer space.

• a pointer to the event associated with the buffer in the static implementation.
• an integer field to specify the last element in the buffer queue. It can either be a head flit or a tail flit.

• a pointer to a process trying to move a packet into the buffer as soon as enough space is available in the buffer. In the dynamic implementation, this process is associated with the head flit of the packet, while in the static implementation, it is associated with the module that currently holds the head flit. This pointer is referred to a pointer to a *waitingpacket*, and is set to NULL when space is available in the buffer.

Input port data structure

The user sets the id and the size of the input port. The fields used inside the simulation routines are

• a pointer to the event associated with the input port in the static implementation.

• a pointer to a semaphore at which user processes wait

• a pointer to a semaphore that controls the movement of data out of the port

Output port data structure

In addition to the id and the size of the output port, the following fields are used inside the simulator.

• a pointer to the event associated with the output port in the static implementation.

• a pointer to semaphore that allows user processes to wait for a packet to arrive at the output port.
- a pointer to a processor connected to the output port in a PARCSIM model.

- a pointer to an event running on the processor in a PARCSIM model.

Arbitration and routing is simulated by modifying the multiplexer and demultiplexer data structures.

4.1.2 Dynamic process implementation

The NewPacket routine creates a packet descriptor and returns a pointer to the packet. The PacketSend routine takes two input arguments, a pointer to the packet and a pointer to an input port, and inserts the packet into the input port queue. In the dynamic implementation, it also creates two processes and assigns them to the packet. One of the processes is associated with the head flit of the packet, and the other is associated with the tail flit of the packet. These processes terminate when the corresponding flits leave the network. Since the processes are implemented as YACSIM events and cannot use local variables, the pointer to the packet data structure is passed as an argument to the two events.

The process associated with the head flit, the HeadEvent, simulates the movement of the head flit. It establishes the path for all the remaining flits of the packet by arbitrating at multiplexers, routing through demultiplexers, waiting until ports are ready or free space is available in buffers, and delaying to simulate the time to move the head flit. The algorithm is shown in Figure 4.3. The process associated with the tail flit, the TailEvent, keeps track of the distribution of the intermediate flits of the packet in the network. It also simulates the movement of the tail flit of the packet, and tears down the path established by the HeadEvent. It schedules the HeadEvents waiting in semaphore queues. Since the HeadEvents are released at the
simulation time when the TailEvents are active, they have to synchronize with the other HeadEvents in the model.

The HeadEvent is initially scheduled by the *PacketSend* routine for the next network cycle. The TailEvent is initially suspended, and is scheduled by the HeadEvent when the head flit moves.

**The HeadEvent**

The first module that the head flit encounters is the input port. If the input port is free, the HeadEvent decrements the size of the port queue and acquires the port semaphore. This semaphore is signaled by the TailEvent associated with the packet after the entire packet moves out of the port. If the port is busy, the HeadEvent waits at the port semaphore.

At a demultiplexer module, the HeadEvent invokes the routing function to determine the output terminal through which the packet will be routed, and delays to simulate the time incurred in routing.

At a multiplexer module, the head flit encounters two sources of delay: the time to perform the arbitration and the time to transfer the head flit through the multiplexer. If the HeadEvent wins the arbitration, it acquires the semaphore. It is released by the corresponding TailEvent after the entire packet moves through the module.

If the semaphore is busy, the HeadEvent waits till the TailEvent of the packet using that channel releases the path. Then, the HeadEvent synchronizes with the other HeadEvents in the system, and simulates the transfer delay through the multiplexer.

**Head flit at a buffer**

When the head flit encounters a buffer, the HeadEvent checks the availability of free flit positions in the buffer. If the number of free flit positions is greater than or
Figure 4.3: Process associated with the head flit
equal to the threshold value for that buffer, the HeadEvent delays to simulate the flit-transfer time, and sets up the path for the remaining flits of the packet. The head flit cannot move if the buffer space does not satisfy the threshold condition. In this case, the waitingpacket pointer is set to point to the HeadEvent and the HeadEvent suspends. It is scheduled when space is made available in the buffer by an event which is moving flits out of that buffer.

After simulating the transfer time, the HeadEvent alters the data structures associated with the packet and the two modules between which the head flit is transferred as shown in the example in Figure 4.4.

![Diagram](image)

(a) Last element in the buffer queue is a head flit
(b) Last element in the buffer queue is a tail flit

**Figure 4.4: Head flit movement from one buffer to the next**

Each buffer holds six flits. At time t=10, a head flit is at the head of the queue in buffer2. The HeadEvent finds that space is available in buffer3 and delays for a cycle to simulate the transfer time. All the other processes are blocked, and none of them wake up until simulation time advances to 11. At t=11, the HeadEvent removes the head flit from the queue of buffer2. It increments the free space for buffer2, moves
the head flit into buffer3 and increments the bubbles for the packet by the amount of available free space, 5 flit-positions. It also schedules its TailEvent to execute after a delta delay.

The HeadEvent also scans the buffer queue to determine the last element in the buffer queue. The last element in the buffer2 queue is a head flit as shown in part a of Figure 4.4. The HeadEvent schedules the TailEvent associated with that packet (the tail flit in buffer1). At time t=11.1, the processes associated with the tail flits in buffer1 and buffer2 are active, and they move the intermediate flits to fill up the bubbles.

The last element in the buffer2 queue is a tail flit as shown in part b of Figure 4.4. The HeadEvent checks the waiting packet pointer of the buffer2 data structure, and schedules the waiting HeadEvent, which is the process associated with the head flit in buffer1.

In Figure 4.4, the head flit moves to the head of queue in buffer3, The HeadEvent remains active and continues to move the head flit forward. However, if the head flit moves into buffer3 behind other flits as shown in Figure 4.5, it gets blocked, and the HeadEvent suspends. It is scheduled by the process associated with the tail flit in front of it in the queue. However, it remains in the blocked state till the head flit gets to the head of the buffer queue. The blocked state of the HeadEvent is described in the next section, along with the description for the TailEvent.

![Figure 4.5: Head flit in a blocked state](image-url)
The head flit leaves the network when it moves into an output port. If the output port is free, the HeadEvent decrements the size of the port queue, acquires the port semaphore, which is released by the corresponding TailEvent when the tail flit moves into the output port, and delays to simulate the transfer time. The HeadEvent sets itself to the delete mode and terminates.

TailEvent

The TailEvent keeps track of the distribution of the intermediate flits and the bubbles of the packet. Initially, the TailEvent is scheduled by the corresponding HeadEvent when the head flit moves and inserts bubbles between the head and the tail flits. The TailEvent moves the flits of the packet to fill up the bubbles, and remains active as long as the bubbles count is non-zero. When the bubble-count is reduced to zero, the tail flit cannot advance any further unless the head flit moves. Therefore, the TailEvent sets the waitingtail pointer of the packet data structure and suspends. It is rescheduled when the HeadEvent advances the head flit further and inserts new bubbles. Figure 4.6 shows the TailEvent algorithm.

Each time the TailEvent moves a flit out of a buffer, it modifies the packet data structure and the data structures of the modules between which the flit is transferred as shown in the example in Figure 4.7.

Each buffer holds seven flits. At time $t=10.1$, a tail flit is at the head of the queue in $buffer_2$. That TailEvent moves a flit from $buffer_2$ to $buffer_3$, and decrements the free space for $buffer_3$. It also scans the queue in $buffer_2$ to determine the last element in that queue before incrementing the free space for $buffer_2$. If the last element is a head flit as shown in part $a$ of Figure 4.7, and if the TailEvent increments the free space in $buffer_2$, it violates the local control protocol. Therefore, it does not
increment the free space. Instead, it schedules the HeadEvent associated with the last head flit.

This HeadEvent, which is in the blocked state, is scheduled at time $t=11.0$, and performs three tasks. It increments the free space in $buffer2$ and increments its bubble count. It also checks its $waitingtail$ pointer, and schedules its TailEvent if that process is suspended.

If the last element in the $buffer2$ queue is a tail flit as shown in part $b$ of Figure 4.7, the TailEvent increments the free space in $buffer2$ directly. Since no other packet is moving into $buffer2$ at that time, local control protocol is not violated. The TailEvent
also checks the *waiting packet* pointer of *buffer2*, and schedules the process waiting to move a packet into *buffer2*.

The tail flit leaves the network when it moves into the output port. The TailEvent releases the port semaphore enabling the output port to receive other packets. It also signals the availability of the packet to the user process associated with the port. It then sets itself to delete mode and terminates.

### 4.1.3 Static process implementation

The module create operations, such as *NewBuffer*, *NewIPort* and *NewOPort*, create a descriptor for the module and return a pointer to the descriptor. In the static process implementation, in addition to allocating memory for the data structures and initializing the various fields, these operations also create a simulation process and assign it to the module. Thus, all the simulation processes are created at the start of the simulation and these processes terminate when the simulation ends. However,
they remain idle until either a head or a tail flit arrives at their module. Since
these processes are implemented as YACSIM events and cannot use local variables, a
pointer to the associated model is passed as an argument to the simulation processes.

**IPortEvent**

The process associated with the input port, the IportEvent, whose algorithm is shown
in Figure 4.8, remains idle until the user process calls the *PacketSend* routine. Then,

![Flowchart](image)

**Figure 4.8: Process associated with an input port**

the process arbitrates at multiplexers, routes the head flit through demultiplexers,
and simulates the movement of the head flit, until it encounters a buffer or an output
port. If space is available in that module, the IPortEvent delays to simulate the
time to transfer the head flit from the input port to the new *headbuffer*. The body
procedure for this part of the IPortEvent routine is similar to the body procedure of
the HeadEvent described in Section 4.1.2.
After moving the head flit out, the IPortEvent reschedules itself to move the remaining flits of the packet. As long as the bubbles count is non-zero, the IPortEvent remains active and moves out a flit on every cycle, and modifies the associated data structures like the TailEvent described in the Section 4.1.2. If the bubble-count is reduced to zero, the IPortEvent suspends and waits for head flit to move and insert new bubbles. When the tail flit leaves the input port, the IPortEvent releases the path between the input port and the new tailbuffer. It then checks the input port queue. If the queue is empty, the IPortEvent suspends. Otherwise, it starts moving the head flit of the packet at the head of the input port queue.

**BufferEvent**

The algorithm for the process associated with a buffer, the BufferEvent, is shown in Figure 4.9. The BufferEvent remains idle until a head flit moves into the buffer. Then,

![Figure 4.9: Process associated with a buffer](image-url)

it simulates the movement of the head flit through multiplexers and demultiplexers
until it encounters a buffer or an output port, the next *headbuffer*. If space is available in the next *headbuffer*, the BufferEvent delays to simulate the transfer time. The body procedure for this part of the BufferEvent routine is similar to the HeadEvent described in Section 4.1.2.

After moving the head flit out, the BufferEvent remains idle until the tail flit moves into the buffer. Then, it remains active as long as the bubble count is non-zero, and moves out a flit on every cycle until the tail flit moves out from the buffer. The body procedure for this part of the BufferEvent routine is similar to the TailEvent described in Section 4.1.2. However, instead of scheduling a blocked HeadEvent, the BufferEvent reschedules itself twice when the last element in the buffer queue is a head flit.

After moving the tail flit out of the buffer queue, the BufferEvent checks the buffer queue. If the queue is empty, it suspends, and waits for a packet to arrive. Otherwise, it starts moving the packet at the head of the buffer queue.

**OPortEvent**

The algorithm for the process associated with the output port, the OPortEvent, is shown in Figure 4.10. The OPortEvent waits for the tail flit of a packet. When it wakes up, it signals the output port semaphore enabling another packet to enter the port queue. It also signals the availability of the packet to the user process associated with the output port.

Thus, in the static implementation, the process associated with a module is active only when either the head or the tail flit is in the module. If a packet is strung across several buffers, only the processes associated with the *headbuffer* and the *tailbuffer* are active.
4.2 Simulating Store and Forward Routing

In store and forward routing, the flits of a packet leave a switch only after the entire packet enters that switch, as discussed in Section 2.3.1. Thus, unlike wormhole routing, the flits of a packet can be spread over only two switches at a time. Figure 4.11 illustrates regular store and forward routing in a network. Each packet consists of five flits and the buffers are large enough to hold one full packet. At time $t=10$, the packet in buffer 2 finds enough space in buffer 3, and starts moving from buffer 2 to buffer 3. At that time, the packet in buffer 1 finds buffer 2 full and cannot move. The packet in buffer 1 starts moving only at time $t=16$ when buffer 2 has enough free space to buffer the entire packet.

Since all the flits of a packet must move into a switch before any of them can move out, a single YACSIM simulation object is sufficient to model the movement of all the flits of the packet. Store and forward routing is implemented in NETSIM using the two different implementation techniques discussed in Section 4.1. The dynamic process implementation is described in Section 4.2.1 and the static process implementation is described in Section 4.2.2. All the simulation processes are implemented as YACSIM events to reduce the memory requirements of the simulator.
4.2.1 Dynamic process implementation

The dynamic implementation of store and forward routing assigns a single simulation process to each packet, unlike the dynamic implementation of wormhole routing, discussed in Section 4.1.2. The PacketSend routine creates this process, referred to as the PacketEvent, and assigns it to the packet. The PacketEvent algorithm is shown in Figure 4.12.

The input port is the first module encountered by a packet. If the port is ready to accept the packet, the PacketEvent decrements the size of the port queue, and acquires the port semaphore. The semaphore is released when the entire packet moves out of the port.

At a demultiplexer, the PacketEvent invokes the routing function associated with the module to determine the output terminal through which the packet will be routed, and delays to simulate the routing time.
Figure 4.12: Process associated with a packet
At a multiplexer, the PacketEvent delays to simulate the time to perform the arbitration and the time to transfer the head flit of the packet through the module. The remaining flits of the packet follow the path established by the head flit.

Packet at a buffer

When the packet encounters a buffer, the PacketEvent checks the availability of free flit positions in the buffer. For store and forward routing, the buffer threshold value is set equal to the packet size. The head flit of a packet can move into a buffer only if the number of free flit positions in the buffer is greater than or equal to the threshold value.

The example in Figure 4.11 shows the movement of the packets in the store and forward routing.

At time $t=10$, the PacketEvent in $buffer_2$ finds enough space in $buffer_3$ for the entire packet. It schedules a move of the head flit of that packet and delays for a cycle to simulate the move time. The PacketEvent associated with the packet in $buffer_1$ is trying to move the packet into a full buffer, $buffer_2$. It suspends after setting the $waitingpacket$ pointer in the $buffer_2$ data structure. At time $t=11$, the PacketEvent in $buffer_2$ moves the head flit descriptor from the queue in $buffer_2$ to the queue in $buffer_3$. It then delays for four cycles to simulate the time to move the remaining four flits of the packet from $buffer_2$ to $buffer_3$. At time $t=15$, the PacketEvent moves the tail flit descriptor from $buffer_2$ to $buffer_3$, and and releases the path established between $buffer_2$ and $buffer_3$. It also checks the $waitingpacket$ pointer associated with $buffer_2$ and schedules the PacketEvent in $buffer_1$.

The output port is the last module that a packet encounters before leaving the network. The PacketEvent releases the port semaphore allowing other packets to use
the port. It also signals the availability of the packet to the user process associated with the port. It then sets itself to delete mode and terminates.

4.2.2 Static process implementation

The static process implementation for store and forward routing is similar to the static process implementation for wormhole routing discussed in Section 4.1.3. It assigns a single simulation process to every buffer and port in the network model. These processes are created and assigned to the module at the start of the simulation and they never terminate. However, in store and forward routing the head flit of a packet cannot leave a module until the tail flit of that packet arrives at that module. Therefore, unlike the processes discussed in Section 4.1.3, these processes remain idle until the tail flit of a packet arrives at their module. Then, they establish a path to the next module that will hold the packet by arbitrating at multiplexers and routing through demultiplexers. Once the path is established, they transfer all the flits to that module. The body procedure of these processes is identical to the PacketEvent described in Section 4.2.1.
Chapter 5

Performance of the Simulator

The performance of the simulator is characterized by simulating NETSIM models in a stand-alone environment. The network load is generated by YACSIM processes associated with each input port. The destination address of the packets generated is uniformly distributed among all the possible destinations. The inter-packet arrival time determines the load of the network, denoted by $\lambda_n$, and is defined as the ratio of the rate at which packets are injected into the network to the bandwidth of the network channel, $B$. A simulation cycle is defined as the time taken to move a flit from one switch to another. In the NETSIM model, a flit is transferred in one simulation cycle; hence the network bandwidth is $B = 1$. The simulation processes associated with the flits that are ready to move are active during a simulation cycle and move the flits from one switch to another. If the YACSIM process injects a packet of length $l$ flits every $T$ simulation cycles, the load $\lambda_n$ is given by $\frac{l}{T}$. The network is driven to saturation when $T$ is equal to $l$.

The performance of the simulator depends on the size of the system being simulated, the routing scheme used in the network, the size of the packets, the size of the network buffers and the packet generation rate. NETSIM models for delta, mesh and hypercube networks were simulated by varying the network size, the routing protocol, the size of the packets, the size of the network buffers and the input load, to characterize the performance of the simulator. Section 5.1 describes the performance of the wormhole routing simulator. Section 5.2 describes the performance of the store and forward routing simulator. Section 5.3 compares the overheads associated with
simulating wormhole routing to the overheads associated with simulating store and forward routing.

5.1 Performance of the wormhole routing simulator

This section discusses the performance of the wormhole routing algorithm described in Section 4.1. NETSIM models for various interconnection networks were simulated on an IBM RS/6000 processor. Section 5.1.1 compares the execution times and the simulation overheads for the dynamic and the static implementation techniques, and Section 5.1.2 analyzes these results.

5.1.1 Comparison of the two implementation techniques

Figure 5.1 shows the ratio of the execution times for the static implementation to the dynamic implementation for simulating a saturated 256-node delta network. The YACSIM process associated with each network input port inserts a packet into the port queue as soon as the port is ready. The X-axis of the graph represents the packet size in flits, which is varied from 4 to 800 flits. The Y-axis represents the ratio of the execution time for the static implementation to the dynamic implementation expressed as a percentage. The relative performance of the two implementation techniques depends on the size of the network buffers. For a network with single-flit buffers, the dynamic implementation is about 15% faster than the static implementation. For a network with a buffer size of four flits, the dynamic implementation is about 20% faster than the static implementation. However, if the buffers are large enough to hold one full packet, the efficiency of the dynamic implementation decreases.
Figure 5.1: Execution time ratio of static to dynamic implementations for a 256-node delta network

Figure 5.2 plots the same ratio for a 256-node mesh network that is driven to saturation. For a network with single-flit buffers, the relative performance of the two implementation techniques is similar to the performance for the delta network. The dynamic implementation is about 15% faster than the static implementation. However, for a mesh network, the efficiency of the dynamic implementation decreases for a buffer size of four flits.

Thus, Figures 5.1 and 5.2 indicate that the efficiency of the dynamic implementation degrades as the buffer size is increased. Results from 64-node network models show a similar trend.

Figure 5.3 shows the performance of the simulator as a function of the applied load for a 256-node delta network. Each node injects a 256-flit packet into the network. The X-axis of the graph represents the input load λ expressed as a percentage. An input load of 5% corresponds to a 256-flit packet injected into the network every 5120 cycles. The Y-axis represents the number of cycles simulated per CPU second, which is the ratio of the total number of simulation cycle to the processor time incurred.
Figure 5.2: Execution time ratio of static to dynamic implementations for a 256-node mesh network

in executing the simulation algorithm, and is a measure of the cost of executing the simulation algorithm.

The dynamic implementation executes more cycles per second than the static implementation for all values of the input load. For single-flit buffers, the dynamic implementation simulates about 250 cycles per second as the input load is increased while the static implementation simulates around 210 cycles per second. For large buffers, the performance degrades rapidly with increase in the input load. For a load of 0.6, the dynamic implementation simulates only 20 cycles per second while the static implementation simulates about 16 cycles per second.

Figure 5.4 shows similar data for a 256-node mesh network. The dynamic implementation is better than the static implementation for networks with single-flit buffers. The performances of the two implementations are almost identical for networks with large flit buffers.
Figure 5.3: Simulator performance as a function of the input load for a 256-node delta network

5.1.2 Analysis of the results

Figures 5.1- 5.4 indicate that the performance of the simulator degrades with increase in the packet size, the buffer size or the input load. For networks with single-flit buffers, the dynamic implementation is about 15% faster than the static implementation. For large buffers, however, the advantage of the dynamic implementation over the static implementation decreases. This section analyzes these results.

Networks with single-flit buffers

In order to determine the reasons for the degradation of the performance of the simulator, the UNIX profiler routine was used to determine the distribution of the overhead among the various simulation routines. The table in Figure 5.5 lists the routines that incurred the largest percentage of the execution time for simulating a 256-node saturated delta network. Each node in the network transmits 250 packets for a total of 64000 packets. The table gives data for a typical short packet and a typical long packet.
Figure 5.4: Simulator performance as a function of the input load for a 256-node mesh network

<table>
<thead>
<tr>
<th>Routines</th>
<th>time spent in the routine in secs (% of the total time)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dynamic Implementation</td>
</tr>
<tr>
<td></td>
<td>pktsize=8</td>
</tr>
<tr>
<td>EventListInsert</td>
<td>103.98 (32.5)</td>
</tr>
<tr>
<td>ShiftBubbles</td>
<td>19.98(6.2)</td>
</tr>
</tbody>
</table>

Figure 5.5: Simulation overheads for a network with single-flit buffers

The EventListInsert and the ShiftBubbles routines account for 40-45% of the total execution time. EventListInsert is the YACSIM routine that puts simulation objects into a time-ordered queue called the EventList, and is invoked each time a simulation process reschedules itself or schedules another process. ShiftBubbles routine is invoked by the process moving the tail flit of a packet to keep track of the distribution of the flits of the packet. In the dynamic implementation, the ShiftBubbles routine is invoked by the TailEvent described in Section 4.1.2 while in the static implementation, the ShiftBubbles routine is invoked by the IPortEvent and the BufferEvent described in Section 4.1.3.
The dynamic implementation is faster than the static implementation because the dynamic implementation schedules fewer processes and incurs a lower overhead in the \textit{EventListInsert} routine as indicated by the data in Figure 5.5. Figure 5.6 shows that the dynamic implementation schedules 512,000 fewer processes than the static implementation. A total of 64,000 packets are sent by the nodes, and the static implementation schedules 8 simulation processes more than the dynamic implementation for each packet, by scheduling one extra process for every stage of the 8-stage 256-node delta network as explained below.

Figure 5.7 shows an example of a network with single-flit buffers. In the dynamic implementation the simulation process is associated with the head flit of the packet, while in the static implementation the simulation processes are associated with the buffers.

In the dynamic implementation, the HeadEvent in \textit{buffer3} finds that it can move the head flit from \textit{buffer3} to \textit{buffer4} at time $t=10$. It delays to simulate the transfer time and reschedules itself. At time $t=11$, the HeadEvent moves the head flit descriptor from \textit{buffer3} to \textit{buffer4}. Since it is already active, it continues to move the head
flit towards the destination module. Thus, the dynamic implementation schedules only one simulation process when a head flit moves from one buffer to another.

In the static implementation, BufferEvent3 finds it can move the head flit at time $t=10.0$. It delays to simulate the transfer time and reschedules itself. At time $t=11$, BufferEvent3 moves the head flit descriptor from buffer3 to buffer4. Although the head flit is now in buffer4, BufferEvent4 is still suspended. Therefore, BufferEvent3 schedules BufferEvent4. Thus, the static implementation schedule two simulation processes when a head flit moves from one buffer to another.

Networks with large buffers

The table in Figure 5.8 lists the overhead incurred in the the EventListInsert and the ShiftBubbles routines for simulating a saturated 256-node delta network with buffers large enough to hold full packet.

<table>
<thead>
<tr>
<th>Routines</th>
<th>time spent in the routine in secs (% of the total time)</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Dynamic Implementation</td>
</tr>
<tr>
<td></td>
<td>pktsize=8</td>
</tr>
<tr>
<td>EventListInsert</td>
<td>606.97 (65.5)</td>
</tr>
<tr>
<td>ShiftBubbles</td>
<td>13.52 (1.5)</td>
</tr>
</tbody>
</table>

Figure 5.8: Simulation overheads for a network with large buffers
As the buffer size is increased, two simulation processes are scheduled to move a flit from one buffer to another as explained in Section 4.1.2. Thus, the performance of the simulator degrades with increase in the size of the network buffers.

The data in Figure 5.8 also indicates that the dynamic implementation incurs a lower overhead in the EventListInsert routine than the static implementation. However, the advantage of the dynamic implementation over the static implementation decreases for networks with large buffers as explained below.

In a network model with large buffers, when a head flit moves into a non-empty buffer behind other flits, the static implementation does not schedule two processes as described as shown in the example in Figure 5.9. At time 10, BufferEvent1 de-

![Figure 5.9: Process scheduling for large buffers](image)

termits it can move the head flit from buffer1 to buffer2 and delays to simulate the transfer time and reschedules itself. At time t=11, BufferEvent1 moves the head flit descriptor from buffer1 to buffer2. The head flit moves into buffer2 behind other flits. Since BufferEvent2 is active moving out the flits of the packet at the head of the queue in buffer2, BufferEvent1 does not schedule BufferEvent2. Thus, the static implementation schedules the same number of processes as the dynamic implementation.
5.2 Performance of the store and forward routing simulator

This section discusses the performance of the store and forward routing simulator described in Section 4.2. Section 5.2.1 compares the execution time for the dynamic and the static implementation techniques and Section 5.2.2 analyzes these results.

5.2.1 Comparison of the two implementation techniques

Simulations similar to those described in Section 5.1.1 are used to evaluate the performance of the store and forward routing simulator.

Figure 5.10 shows the ratio of the execution times for the static to the dynamic implementation techniques for saturated 256-node networks with buffers large enough to hold one full packet. The X-axis of the graph represents the packet size in flits. The Y-axis is the ratio of the execution time for the static implementation to the dynamic implementation, with a ratio greater than 1 indicating that the dynamic implementation is better than the static implementation. The performance of the
two implementation techniques is identical for both the delta network model as well as for the mesh network model.

Figure 5.11 shows the performance of the simulator as a function of the applied load for 256-node networks. Each node injects a 256-flit packet into the network.

![Graph showing simulator performance](image)

**Figure 5.11: Simulator performance in cycles per second for 256-node network models**

The X-axis of the graph represents the input load $\lambda$ expressed as a percentage. The Y-axis represents the number of cycles simulated per CPU second, which is the ratio of the total number of simulation cycle to the processor time incurred in executing the simulation algorithm.

The performance of the two implementation techniques is identical for various input loads.

### 5.2.2 Analysis of the results

In store and forward routing, the head flit of a packet is allowed to move out of a buffer only after the tail flit of that packet has arrived at that buffer. Therefore, unlike the wormhole routing simulator, the dynamic implementation of store and forward routing
simulator does not have any advantage over the static implementation. Thus, the performance of the two implementation techniques for simulating store and forward routing is almost identical.

5.3 Comparison of simulation overheads for wormhole and store and forward routing

In wormhole routing, the flits of a packet can move asynchronously, and this asynchronous movement must be modeled to develop an accurate simulator. Therefore, the performance of the wormhole routing simulator degrades rapidly with increase in the size of the packets, the size of the buffers and the load.

Store and forward routing allows the flits of a packet to get spread-out over at most two switches. This simplifies the simulation model and the performance of the store and forward routing simulator is less sensitive to variations in the packet size.

Figure 5.12 compares the time taken to simulate a 256-node delta network using wormhole and store and forward routing. The X-axis of the graph represents the packet size in flits. The Y-axis represents the total execution time in seconds. The

![Graph comparing execution times for wormhole and store and forward routing]

Figure 5.12: Comparison of execution times for simulating a 256-node delta network
dynamic implementation of wormhole routing is used in the comparison since it is faster than the static implementation. The overhead incurred in simulating wormhole routing with large buffers is about 10 times the overhead incurred in simulating store and forward. The overhead of simulating store and forward increases at the rate of 5 seconds per flit in the packet while the overhead of simulating wormhole routing increases at the rate of 50 seconds per flit in the packet.

Figure 5.13 compares the performance of wormhole and store and forward routing simulators as a function of the input load for 256-node networks. For a lightly loaded network, the overhead in simulating wormhole routing is about twice the overhead in simulating store and forward. However, as the load increases, the overhead in simulating wormhole routing is ten times the overhead in simulating store and forward.

![Figure 5.13: Comparison of simulation rates for simulating 256-node networks](image)

Thus, simulating wormhole routing in detail is expensive. Chapter 6 discusses techniques to reduce the overhead incurred in simulating wormhole routing.
Chapter 6

Approximate Models for Simulating Wormhole Routing

The results in Chapter 5 indicate that the overhead incurred in simulating wormhole routing is about ten times the overhead incurred in simulating store and forward routing. The performance of the wormhole routing simulator degrades with increases in the packet size, the buffer size and/or the network load.

This chapter describes four techniques that approximate the behavior of wormhole routing. These techniques do not model the movement of the flits of a packet in detail. Therefore, they improve the performance of the simulator and make the simulation overhead independent of the size of the packet. However, they introduce errors in the network performance metrics predicted by the model. The trade-off between speedup and accuracy of the approximate models is also discussed in this chapter.

6.1 Techniques to reduce simulation overhead

The data in Figures 5.5 and 5.8 indicate that for networks with single-flit buffers about 45% of the execution time is incurred by the EventListInsert and the ShiftBubbles routines while for networks with large buffers the overhead incurred by these routines is 50 to 70% of the total execution time.

The overhead incurred in simulating wormhole routing can be reduced by applying techniques used to simulate store and forward routing to model wormhole routing. However, the simulation model for store and forward routing cannot be directly used to approximate wormhole routing. Figure 6.1 shows that the latency for packets using store and forward routing is almost twice the latency incurred in a network
Figure 6.1: Comparison of latency estimates for a 256-node delta network

using wormhole routing. Therefore, approximating wormhole routing with store and forward would result in errors close to 100% in the latencies predicted by the model.

The approximate techniques described in this chapter modify the simulation algorithm for wormhole routing described in Chapter 4. The technique described in Section 6.2 reduces the simulation overhead by reducing the total number of processes scheduled during the simulation. Section 6.3 describes a technique that reduces the simulation overhead by cutting down the bubble-management overhead and by scheduling fewer processes. Section 6.4 describes a technique that combines the techniques presented in Sections 6.2 and 6.3. Section 6.5 describes another technique that further reduces the number of processes scheduled during the simulation. Section 6.6 compares the approximate models and evaluates the trade-off between accuracy and speed for the four approximate models.
6.2 Buffer Approximation

This section describes a simulation algorithm for wormhole routing that approximates the movement of the flits inside a buffer. This technique is applicable only to networks with multiple flit buffers, and is designed to decrease the simulation overhead by reducing the total number of processes scheduled.

In the detailed model for wormhole routing, each time a flit moves out of a full buffer two simulation processes are scheduled at a delta offset from each other, as explained in Section 4.1.2, in order to ensure that the switch uses local control to transfer flits.

In the approximate model, each time a flit moves out of a full buffer only one simulation process is scheduled as shown in Figure 6.2. Unlike the detailed model, the process associated with the tail flit of the packet that is moving out does not schedule the process associated with the blocked head flit at the end of the buffer.
queue. The TailEvent moves the flit out of the buffer and increments the free space in the buffer. Thus, the total number of simulation processes scheduled per buffer is reduced by half.

Buffer approximation assumes that the status of free space in buffers is known globally across several buffers. Therefore, flits move in and out of buffers at the same simulation time, and violate the local control protocol for flit transfer. Since the flits move at an earlier simulation time, buffer approximation predicts lower values for latency compared to the detailed model.

Buffer approximation reduces the total number of processes scheduled for each buffer by half only if that buffer is large enough to hold at least a tail flit and the head flit of another packet. Therefore, it does not speed up the simulation of network models with single-flit buffers.

6.2.1 Evaluation of the buffer approximation technique

The performance of the buffer approximation technique is evaluated by simulating NETSIM models for various networks in a stand-alone environment on an IBM RS/6000 processor.

Figure 6.3 shows the speedup obtained by the buffer approximation technique for 256-node network models with a buffer size of 4 flits. The Y-axis of the graph represents the ratio of the execution time for the detailed technique to the execution time for buffer approximation. The X-axis represents the packet size in flits. The speedup obtained for a network model with a buffer size of 4 flits is close to one. Thus, buffer approximation does not offer any significant advantage for simulating networks with small buffers.

Buffer approximation benefits networks with large buffers as it reduces the number of processes scheduled by half. Figure 6.4 shows the speedup obtained for 64-node
network models with buffers large enough to hold one full packet. The network is driven to saturation with the process associated with each input port inserting a packet into the network as soon as the port is ready. For a delta network, buffer approximation is about 1.9 times faster than the detailed model while for a mesh network model it is about 2.2 times faster.

Figure 6.5 shows the error in the latency values predicted by the simulator for the corresponding network models. The error is expressed as a percentage and plotted on the Y-axis. The simulator predicts lower values of latency resulting in negative

Figure 6.4: Speedup for 64-node networks with large buffers
Figure 6.5: Error in latency for 64-node networks with large buffers

errors. Buffer approximation introduces an error of 0.2% for delta network and an error of 0.8% for mesh networks.

Figure 6.6 shows the speedup obtained for 256-node network models with buffers large enough to hold one full packet. The network is driven to saturation. For a delta network, buffer approximation is about 4.5 times faster than the detailed model. The speedup for a mesh network is about 3 while the speedup for a binary n-cube network is 2.

Figure 6.6: Speedup for 256-node networks with large buffers
Buffer approximation offers greater advantage to the delta network because all the packets in the network move through all the stages of the network. The data in Figure 6.7 shows the reduction in the process-scheduling overhead.

<table>
<thead>
<tr>
<th>Routines</th>
<th>% of the total time spent in the routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Detailed Model</td>
</tr>
<tr>
<td></td>
<td>pktsize=8</td>
</tr>
<tr>
<td>EventListInsert</td>
<td>65.5</td>
</tr>
</tbody>
</table>

**Figure 6.7**: Simulation overheads for a 256-node delta network with large buffers

The benefits of buffer approximation for a direct network depend on the lengths of paths in the network, and it offers greater advantages for longer paths with more buffers.

For uniformly distributed traffic in a \(k\)-ary \(n\)-cube direct network of dimension \(n\) with \(k\) nodes in each dimension and with no end-around connections, the average path length is given by Equation 6.1 [2].

\[
n(k - 1/k)/3
\]

For a 64-node mesh network, \(n=2\) and \(k=8\), and the average path length is 5.25 hops while for a 256-node mesh network, \(n=2\) and \(k=16\), and the average path length is 11 hops. For a 64-node hypercube network, \(n=6\) and \(k=2\), and the average path length is 3 hops while for a 256-node hypercube, \(n=8\) and \(k=2\), and the average path length is 4 hops.

The length of the average path and the longest path is greater in a mesh network than in a hypercube with the same number of nodes. Therefore, buffer approximation offers better speedup for mesh networks.

The error introduced by buffer approximation for the 256-node networks is shown in Figure 6.8. The errors are negative since buffer approximation predicts lower values
Figure 6.8: Error in latency for 256-node networks with large buffers

for latency. The error for a delta network is about 1.2% while the errors for the mesh network and the hypercube network are within 0.4%.

Figure 6.9 shows the performance of buffer approximation as a function of the input load for 256-node networks. The process associated with each input port inserts a 256-flit packet into the network. The X-axis represents the input load in percentage of the saturated load. The Y-axis is the ratio of the number of simulation cycles executed per second by the buffer approximation model to the number of cycles executed per second by the detailed model.

As the load increases, contention in the network increases and packets get spread out over several buffers. Since buffer approximation reduces the overhead involved in simulating full buffers, the performance of the buffer approximation improves with increase in load.

In summary, buffer approximation achieves speedups of up to 4.5 over the detailed model for networks with large buffers. The speedup increases with the size of the network, the path length and the network load. The accuracy of the buffer approximation model is within 1.2% of the detailed model.
Figure 6.9: Performance of buffer approximation as a function of the input load

6.3 Tail-flit Approximation

This technique approximates the movement of the tail flit of a packet after the head flit of that packet leaves the network. It models the movement of the tail flit by techniques similar to those used for simulating store and forward routing. The simulation overhead is decreased by reducing the bubble-management overhead and by scheduling fewer simulation processes.

The movement of a packet is modeled accurately until the head flit of the packet leaves the network. Once the head flit leaves the network, the tail flit does not encounter any further delay due to contention. Hence, the time taken by the tail flit to leave each intermediate buffer can be estimated. Therefore, instead of rescheduling on every cycle and moving one flit at a time from a module, the TailEvent schedules itself once for each port or buffer in the path and moves out all the flits of the packet from that module, reducing the bubble-management and process-scheduling overheads.

Tail-flit approximation introduces two sources of errors in the latency predicted for the packets. It assumes that the tail flit can move forward on every cycle after
the head flit leaves the network. If the bubble distribution at the time the head flit leaves the network allows the tail flit to advance on every subsequent cycle, no error is introduced by this model. However, if the bubble distribution does not allow the tail flit to move forward on every subsequent cycle, the approximate technique introduces an error in the time at which the tail flit clears the path and releases the channels, as shown in Figure 6.10. The approximate model assumes that the availability of free

![Diagram of Detailed and Approximate Models](image)

**Figure 6.10: Tail-flit approximation with global routing control**
	space is signaled globally and violates the local control protocol for flit transfer. As a result, the TailEvent releases the channels earlier than the correct simulation time and the model gives lower estimates for the network latency.

The other error introduced by this approximate model manifests itself if the tail flit of the packet is in a buffer at the time the head flit leaves the network as shown in Figure 6.11.

At time $t=11$, a head flit moves from $buffer2$ to the output port. The tail flit of that packet is in $buffer1$. In the approximate model, the process associated with that
Figure 6.11: Tail-flit approximation in a buffer

tail flit schedules itself only once to move all the four flits from buffer1 to buffer2 instead of scheduling itself four times. At t=14.1, the process transfers the tail flit from buffer1 to buffer2. It also signals the availability of free space in buffer1. As a result, the flits of the packet moving into buffer1 move at a later simulation time.
At t=15, four flits of the packet move into buffer1 in the detailed model while only one flit moves into that buffer in the approximate model. Thus, the model predicts a higher latency for that packet.

The overall error introduced by tail-flit approximation depends on the size of the packet and the length of the path from the source to the destination.

6.3.1 Evaluation of the tail-flit approximation technique

NETSIM models for various networks were simulated in a stand-alone environment on an IBM RS/6000 processor to evaluate the performance of the tail-flit approximation technique.
Figure 6.12 shows the speedup of the approximate model for 64-node networks with single-flit buffers. The X-axis represents the packet size in flits. The Y-axis represents the ratio of the execution time for the detailed technique to the execution time for the tail-flit approximation. The network is driven to saturation and each node inserts 1000 packets into the network. Unlike buffer approximation discussed in Section 6.2.1, tail-flit approximation offers a tremendous advantage for networks with single-flit buffers. The simulation overhead is almost independent of the size of the packet resulting in linear speedups for all the three network models.

Figure 6.13 shows the error in the latency predicted by the simulator for the 64-node network models with single-flit buffers. The errors are within 0.65%.

Figure 6.14 shows the speedup of the approximate model for 64-node networks with buffers large enough to hold one full packet. The X-axis represents the packet size in flits and the Y-axis represents the speedup. The speedup obtained is similar to the speedup for buffer approximation.

Figure 6.15 shows the error introduced by this approximation for networks with large buffers. Due to the short path lengths in the delta and the hypercube networks, the tail flit is in the input port when the head flit leaves the network. As a result,
the tail flit approximation predicts lower latencies resulting in negative errors. In a mesh network, the error is positive for packets that are short enough such that the tail flit is in a buffer when the head flit leaves the network. However, for long packets the errors are negative.

Figure 6.16 shows the performance of tail-flit approximation for 256-node networks with single-flit buffers. The network is driven to saturation, and each node inserts 250 packets into the network. Tail-flit approximation makes the simulation overhead independent of the packet size resulting in linear speedup.
Figure 6.15: Error in latency for 64-node networks with large buffers

Figure 6.16: Speedup for 256-node networks with single-flit buffers

Figure 6.17 shows the error in latency predictions for the 256-node networks with single-flit buffers. The errors for the 256-node networks are similar to the errors for the 64-node networks shown in Figure 6.13, and are within 0.6%.

Figure 6.18 shows the speedup of tail-flit approximation for 256-node networks with buffers large enough to hold one full packet. The approximate models is up to three times faster than the detailed model.

The errors in the predicted latencies shown in Figure 6.19 indicate the following characteristics. The errors depend on the packet size, the length of the path and the location of the tail flit when the head flit leaves the network. Figure 6.20 tabulates
Figure 6.17: Error in latency for 256-node networks with single-flit buffers

Figure 6.18: Speedup for 256-node networks with large buffers

the location of the tail flit when the head flit leaves the network. In a hypercube, the path lengths are small. Therefore, the tail flit of a third of all the longer packets is in the input port when the head flit leaves the network resulting in positive errors for packets smaller than 8 flits and negative error for all other packet sizes. For delta networks, the crossover point of the error curve shifts further to the right. The model predicts higher latencies for packets up to 80 flits long. The mesh network has the longest path length among the three networks. Therefore, most of the packets are in the network when the head flit leaves the network. This increases the predicted
Figure 6.19: Error in latency for 256-node networks with large buffers

Figure 6.20: Location of the tail flit when the head flit leaves the network for networks with large buffers

<table>
<thead>
<tr>
<th>Packetsize</th>
<th>Count of number of packets</th>
<th>Mesh Network</th>
<th>Hypercube network</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>input port</td>
<td>buffer</td>
<td>input port</td>
</tr>
<tr>
<td>8</td>
<td>144</td>
<td>63856</td>
<td>13548</td>
</tr>
<tr>
<td>32</td>
<td>1029</td>
<td>62971</td>
<td>20698</td>
</tr>
<tr>
<td>320</td>
<td>2010</td>
<td>61990</td>
<td>23751</td>
</tr>
<tr>
<td>400</td>
<td>2013</td>
<td>61987</td>
<td>23846</td>
</tr>
</tbody>
</table>

Latency values and cancels out the effect of predicting lower values of latency for long packets.

Figure 6.21 shows the performance of tail-flit approximation as a function of the input load for 256-node networks. The X-axis represents the input load and the Y-axis represents the ratio of the number of simulation cycles executed per second by the approximate model to the number of cycles executed per second by the detailed model. For networks with single-flit buffers, the approximate model executes 5 to 7 times more cycles than the detailed technique. For networks with large buffers, the ratio ranges from 1.6 to 2.5.
Figure 6.21: Performance of tail-flit approximation as a function of the input load

6.3.2 Analysis of the results

Networks with single-flit buffers

The graphs in Section 6.3.1 indicate that for networks with small flit buffers, tail-flit approximation makes the simulation overhead independent of the packet size and introduces less than 0.8% error in the latency values, thus offering a tremendous saving in the execution time with very little loss in accuracy.

The table in Figure 6.22 compares the simulation overheads of tail-flit approximation model to the detailed model for a 256-node delta network. For a packet size of 8 flits, reduction in simulation overhead is not significant. However, for a long packet,
tail-flit approximation cuts down the overhead incurred in the `EventListInsert` routine by more than half and the overhead incurred in the `ShiftBubbles` routine by a factor of 6. In the approximate model, the bubble-management overhead is independent of the packet size and accounts for the linear speedup obtained by the tail-flit approximation model.

Networks with large buffers

For networks with large buffers, tail-flit approximation offers a speedup of up to 3 with errors up to 8%. The table in Figure 6.23 indicates that the overhead incurred in the `ShiftBubbles` routine is a small percentage of the total execution time. Since the tail-flit approximation technique primarily reduces the bubble-management overhead, the speedups obtained for networks with large buffers is lower compared to the speedups obtained for networks with small buffers.

<table>
<thead>
<tr>
<th>Routines</th>
<th>% of the total time spent in the routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Detailed Model</td>
</tr>
<tr>
<td></td>
<td>pktsize=8</td>
</tr>
<tr>
<td>EventListInsert</td>
<td>65.5</td>
</tr>
<tr>
<td>ShiftBubbles</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Figure 6.23: Simulation overheads for a 256-node delta network with large buffers

6.4 Combined Approximate Model

This section presents the results for an approximate model that combines buffer approximation described in Section 6.2 and tail-flit approximation described in Section 6.3. Since, buffer approximation is not applicable to networks with small flit buffers, the combined model is the identical to tail-flit approximation for such networks.
For networks with large buffers, the combined model approximates the movement of the flits inside a full buffer and approximates the movement of the tail flit after the head flit leaves the network. Therefore, it reduces the process-scheduling and bubble-management overheads in a single simulation model. Section 6.4.1 presents the results for this approximate model and Section 6.4.2 presents an analysis of the results.

6.4.1 Evaluation of the combined approximate model

Experiments similar to those described in Section 6.3.1 are used to evaluate the performance of the combined approximate model. Network models with single-flit buffers are not evaluated as the combined approximate model is identical to the tail-flit approximation model for such networks.

Figure 6.24 shows the speedup of the combined approximate model for 256-node networks. The X-axis represents the packet size in flits and the Y-axis represents the speedup obtained by the combined approximate model. The network is driven to saturation and each node inserts 250 packets into the network. The speedups range from 3 to 5.

Figure 6.24: Speedup for 256-node networks with large buffers
Figure 6.25 shows the error in latency for the 256-node networks. In a mesh network, the path lengths are longer. Therefore, for short tail-flit approximation predicts higher values of latency resulting in positive errors. However, as the packet size increases the negative errors due to the buffer approximation dominate resulting in negative errors overall. The errors for the delta and hypercube networks are also negative since both buffer approximation and tail-flit approximation predict lower values for latency.

Figure 6.26 shows the performance of the combined approximate model as a function of the input load for 256-node networks. The X-axis represents the input load and the Y-axis represents the ratio of the number of simulation cycles executed per second by the approximate model to the number of cycles executed per second by the detailed model. Contention in the network increases with the load. Therefore, the performance of the combined approximate model improves with increase in load.
6.4.2 Analysis of the results

Figure 6.27 tabulates the simulation overheads of the approximate model and the detailed model for a 256-node delta. The combined approximate model reduces the overhead incurred in the EventListInsert and the ShiftBubbles routines significantly.

<table>
<thead>
<tr>
<th>Routines</th>
<th>% of the total time spent in the routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Detailed Model</td>
</tr>
<tr>
<td></td>
<td>pktsize=8</td>
</tr>
<tr>
<td>EventListInsert</td>
<td>65.5</td>
</tr>
<tr>
<td>ShiftBubbles</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Figure 6.27: Simulation Overheads for a 256-node delta network with large buffers

6.5 Extension of the combined model

This section presents an extension to the combined approximate model described in Section 6.4. In this model, the HeadEvent and the TailEvent associated with a packet are scheduled at the same simulation time instead of being scheduled at an offset from each other.
In the detailed model the offset is used for local control protocol. A HeadEvent waiting at a semaphore is always released from the semaphore queue by a TailEvent. The HeadEvent reschedules itself to synchronize with the other HeadEvents in the system as shown in Figure 4.3. The extended model eliminates this offset and reduces the process synchronization overhead. As a result, the process-scheduling overhead is further reduced. Since the elimination of the offset violates local control, the extended approximation model predicts lower latencies.

Section 6.5.1 presents the results for the extended approximate model and Section 6.5.2 presents an analysis of the results.

6.5.1 Evaluation of the extended approximate model

The extended approximate model is characterized by experiments similar to those described in Section 6.4.1

Figure 6.28 shows the speedup of the extended approximate model for 256-node networks with single-flit buffers. The network is driven to saturation and each node inserts 250 packets into the network. The extended approximate model achieves linear speedup and makes the simulation overhead independent of the packet size.

![Figure 6.28: Speedup for 256-node networks with single-flit buffers](image-url)
Figure 6.29 shows the latency error for 256-node network models with single-flit buffers. The extended approximate model predicts lower values for latency resulting in negative errors.

![Graph showing latency error for different network models](image)

**Figure 6.29: Error in latency for 256-node networks with single-flit buffers**

Figure 6.30 shows the speedup of the extended approximate model for 256-node networks with large buffers and Figure 6.31 shows the latency error introduced by this approximation. The negative errors introduced by buffer approximation and by the elimination of the offset dominate the overall errors.

![Graph showing speedup for different network models](image)

**Figure 6.30: Speedup for 256-node networks with large buffers**
Figure 6.31: Error in latency for 256-node networks with large buffers

Figure 6.32 shows the performance of the extended approximation as a function of the input load for 256-node network models. The X-axis represents the input load and the Y-axis represents the ratio of the number of simulation cycles executed per second by the approximate technique to the number of cycles executed per second by the detailed technique. For networks with single-flit buffers, the approximate technique executes 5 to 8 times more cycles than the detailed technique while for networks with large buffers the ratio ranges from 2 to 5.
6.5.2 Analysis of the results

The data in Figure 6.33 compares the simulation overheads of the extended approximation model to the combined approximation model for a 256-node delta network with single-flit buffers.

<table>
<thead>
<tr>
<th>Routines</th>
<th>% of the total time spent in the routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extended Approximation</td>
</tr>
<tr>
<td></td>
<td>pktsize=8</td>
</tr>
<tr>
<td>EventListInsert</td>
<td>8.9</td>
</tr>
<tr>
<td>ShiftBubbles</td>
<td>5.0</td>
</tr>
</tbody>
</table>

Figure 6.33: Simulation overheads for a 256-node delta network with single-flit buffers

Figure 6.34 compares the overheads of the extended approximation model to the combined approximation model for a 256-node delta network with large buffers.

<table>
<thead>
<tr>
<th>Routines</th>
<th>% of the total time spent in the routine</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Extended Approximation</td>
</tr>
<tr>
<td></td>
<td>pktsize=8</td>
</tr>
<tr>
<td>EventListInsert</td>
<td>8.6</td>
</tr>
<tr>
<td>ShiftBubbles</td>
<td>2.4</td>
</tr>
</tbody>
</table>

Figure 6.34: Simulation overheads for a 256-node delta network with large buffers

6.6 Comparison of the approximate models

This section compares the four approximate models and evaluates the trade-off between accuracy and speedup for each model.
6.6.1 Approximate models for delta networks with small buffers

Buffer approximation is not applicable to networks with small buffers. Therefore, the combined approximation model is identical to the tail-flit approximation model. Figure 6.35 compares the speedups obtained by the tail-flit approximation and the extended approximation models for a 256-node delta network. The extended approximation model is about 40% faster than the tail-flit approximation model for all packet sizes as it reduces the process-synchronization overhead.

Figure 6.36 shows that the extended model does not introduce any further errors in the latency values predicted by the tail-flit approximation model. Therefore, the extended approximation model offers the best compromise between speedup and accuracy for a 256-node delta network with small buffers.

6.6.2 Approximate models for delta networks with large buffers

Figure 6.37 compares the speedups and Figure 6.38 compares the errors introduced by each approximate model. For networks with large buffers, the process-scheduling overhead is a greater percentage of the overall simulation overhead. Buffer approximation reduces this overhead and performs better than tail-flit approximation. Buffer
Figure 6.36: Error in latency for 256-node delta network with single-flit buffers

Figure 6.37: Speedup for 256-node delta network with large buffers

approximation always predicts low and the errors are within 1.2%. The errors introduced by tail-flit approximation depend on the ratio of the length of the packet to the length of the path and are within 3%. Thus, buffer approximation provides greater speedup with lower errors. The combined model is faster than the buffer approximation model but the errors are higher (within 6%). The extended approximation model is faster than the combined model for small packets but has no additional advantage over the combined model for large packets.
Figure 6.38: Error in latency for 256-node delta network with large buffers

Thus, for delta networks with large buffers, buffer approximation provides the best compromise between speedup and error with speedups up to 4.5 and errors within 1.2%.

6.6.3 Approximate models for mesh networks with small buffers

Figure 6.39 compares the speedups obtained by the tail-flit and the extended approximation models for a 256-node mesh network. Figure 6.40 compares the errors introduced by each of the two models. The speedups obtained by the two techniques are almost identical. The errors introduced by the two models are also equal in magnitude. Thus, for mesh networks the performance of the two techniques is almost identical.

6.6.4 Approximate models for mesh network with large buffers

Figure 6.41 compares the speedups of the four approximate models for a 256-node mesh network and Figure 6.42 shows the errors introduced by the models. Buffer approximation model is faster than tail-flit approximation by a factor of 1.8—while the
errors for both the models are almost identical and within 1%. Although the combined approximation model is about 1.2 times faster than the buffer approximation model, it introduces greater errors. The performance of the extended approximation model is identical to the performance of the combined approximation model.

Thus, for mesh networks with large buffers, buffer approximation provides the best compromise between speedup and accuracy with speedups up to 3 and errors less than 1%.
Figure 6.41: Speedup for 256-node mesh network with large buffers

Figure 6.42: Error in latency for 256-node mesh network with large buffers

6.6.5 Approximate models for cube networks with small buffers

Figure 6.43 compares the speedups obtained by the tail-flit and the extended approximation models for a 256-node cube network. Figure 6.44 shows the errors introduced by each of the two approximate models. The extended approximation model is about 12% faster than the tail-flit approximation model for large packets. The errors introduced by the two models are almost identical and within 0.4%.
Figure 6.43: Speedup for 256-node cube network with single-flit buffers

Figure 6.44: Error in latency for 256-node cube network with single-flit buffers

6.6.6 Approximate models for cube networks with large buffers

Figure 6.45 compares the speedup of the four approximate models for a 256-node cube network and Figure 6.46 shows the errors introduced by the models. Unlike the results discussed in sections 6.6.2 and 6.6.4, for a hypercube network, tail-flit approximation is 20% faster than buffer approximation. The average path in a hypercube network are much shorter than in any of the other networks, and the packets traverse fewer buffers. As a result, buffer approximation does not achieve comparable speedups However, the errors introduced by tail-flit approximation are significantly
higher than the error introduced by buffer approximation. The combined and the extended approximation models are almost twice as fast as the buffer approximation model, but introduce greater errors. Thus, the buffer approximation model still provides the best compromise between speedup and accuracy.

6.6.7 Overview of the results

The results in Sections 6.6.1 to 6.6.6 indicate that the best approximate model for a network depends on the buffer size, the packet size and the load. For networks with
small buffers, the extended model provides the best compromise between speedup and accuracy, irrespective of the packet size and the load.

For networks with large buffers, the choice of the best approximate model depends on the packet size and the load. Figure 6.47 shows the best approximate model under different conditions. For a heavily loaded network, buffer approximation provides the best compromise between speedup and accuracy irrespective of the packet size. However, buffer approximation offers no significant advantage in a contention-free network. Therefore, in a lightly loaded network, the extended model provides the best compromise between speedup and accuracy for small packets and the combined model is the best for large packets.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Light load</th>
<th>Heavy load</th>
</tr>
</thead>
<tbody>
<tr>
<td>Small packets</td>
<td>Extended</td>
<td>Buffer</td>
</tr>
<tr>
<td>Large packets</td>
<td>Combined</td>
<td>Buffer</td>
</tr>
</tbody>
</table>

Figure 6.47: Best approximate model for networks with large buffers
Chapter 7

Simulation Model for the iPSC/860 Network

Validation of a simulation model is necessary to gain confidence in the accuracy of its predictions. This chapter describes the approach used to design validation experiments for a NETSIM simulation model.

A NETSIM model for the iPSC/860 interconnection network is used to validate the simulator. Estimates of the network latency predicted by the simulation model are compared with the latencies measured on the iPSC/860 multicomputer, to verify the accuracy of the model.

Sections 7.1 and 7.2 describe the architecture and the message passing protocols of the iPSC/860 network. Section 7.3 presents the NETSIM model developed for the iPSC/860 network and Section 7.4 evaluates the performance of the NETSIM model.

7.1 iPSC/860 Network Architecture

Intel's personal supercomputer, the iPSC/860, is an i860 processor-based message passing concurrent system. The processing nodes of the iPSC/860 are connected to Direct-Connect routers [27]. The Direct-Connect routers are interconnected by bit-serial, full-duplex channels to form a hypercube network. Each Direct-Connect router (also called a Direct-Connect Module or DCM) supports eight channels, and can be interconnected to form networks of up to seven dimensions with 128 processing nodes. Figure 7.1 shows a 3-dimensional hypercube network with the channel and node numbering convention.

In a hypercube of dimension 3, each node has a direct connection to three other nodes, called the nearest neighbor nodes. The nodes are assigned unique addresses
such that the addresses of any two nearest neighbor nodes differ by one bit. The channel connecting two nearest neighbors is numbered according to its dimension. If the addresses of the two nearest neighbors differ in bit position \(i\), the channel connecting the two nodes is referred to as channel \(i\) or the channel in the \(i\)-th dimension. For example, in Figure 7.1, the addresses of nodes 0 and 1 differ in bit position 0. Therefore, the channel connecting nodes 0 and 1 is referred to as channel 0.

7.1.1 Router Architecture

The internal organization of the DCM is a full crossbar. It has eight independent routing elements, one for each channel, allowing up to eight simultaneous message transmissions through the DCM. Each routing element can drive any one of the outgoing channels. Since more than one routing element may request the same output channel, an arbitration mechanism is provided for each output channel.

The processing node is connected to the DCM through two unidirectional, parallel buses, the node-source channel which puts data into the network and the node-sink channel which removes data from the network.
In addition to establishing a data path through the DCM, the routers and arbiters associated with the channel also establish a status path in the opposite direction by setting the status switch. The status path routes status information necessary to provide flow control for messages.

7.1.2 Channel Architecture

The bit-serial, full-duplex channel between two DCM's physically consists of four conductors as shown in Figure 7.2. Each pair of conductors operates independently,

![Diagram of Channel Configuration](image)

**Figure 7.2: Channel configuration**

and is driven in only one direction. Serial data, control bits and status bits are transferred across the data lines. The strobe lines are used to validate the data lines and provide the clock source for the next router, as the whole system is synchronous.

7.2 Message Passing Protocols

The switching protocol used in the iPSC/860 is a variation of wormhole routing described in Section 2.3.2. In Direct-Connect routing, the message is transmitted only after the entire path between the source and destination nodes is established by a header.
7.2.1 Routing Algorithm

The path taken by a message between any two nodes in the system is unique and is determined by the e-cube routing algorithm [17] [34]. The e-cube algorithm guarantees that no circularities will occur in message routing and thus prevents hardware deadlock. The e-cube routing algorithm states that messages are routed in increasingly higher channel dimensions until the destination is reached, with the channels being labeled according to the numbering convention shown in Figure 7.1. Figure 7.3 shows the path for a message from node 0 to node 7. It consists of channel 0, channel 1 and channel 2, and passes through the routers at nodes 1 and 3. However, a message from node 7 to node 0 does not take the same path since that would violate the e-cube algorithm. Instead, it is routed along increasing channel dimensions through the routers at nodes 6 and 4.

7.2.2 Routing Operation

A routing operation consists of four phases: establishment the path, acknowledgement, message transmission and release of the channels.
Establishing the path

The source node initiates the message transfer by forwarding, at a minimum, one 32-bit word to the DCM. This 32-bit word includes the routing probe that contains the addressing information and is used to establish the path. The routing probe is computed by taking the “exclusive or” of the binary address of the source and the destination nodes. Each bit of the routing probe corresponds to a channel on which the message can be routed. If a bit is set, the message is routed along that channel. A multi-hop message is one that is transmitted along two or more channels and passes through intermediate routers.

The node-source channel requests the outgoing channel corresponding to the lowest order bit set in the routing probe. The local DCM uses a round-robin arbitration scheme to arbitrate requests for that channel. The routing probe is then transmitted to the next DCM along that channel.

A DCM receiving the routing probe determines the next lowest order bit set in the routing probe. If none of the bits are set, the routing probe is at the destination node. Otherwise, the intermediate DCM requests the outgoing channel corresponding to the lowest order bit set in the routing probe. The routing probe arbitrates for the next channel. If the requested output channel is busy, it is buffered in the intermediate DCM. Thus, the routing probe retains the path that it has already acquired while waiting to establish the remainder of the path.

The routing probe is forwarded until it reaches the destination node DCM. When the destination node receives the routing probe, the message path is completely established. This path is retained till the entire message is transmitted. If the destination DCM is ready to accept the message, it initiates the acknowledgement phase.
Acknowledgement phase

The acknowledgement phase requires that a deterministic connection be made from the destination DCM back to the source DCM. This path carries the flow control information and is termed as the ‘status path’. The status path follows the message path but in the opposite direction, from the destination to the source. The status path is established as a result of forming the message path. The example in Figure 7.4 shows the acknowledgement phase for a message from node 2 to message 1. The status path is also retained for the entire duration of the message transmission. The destination node transmits a RDY bit along the status path which traces its way back to the source node.

Message transmission

When the source node receives the RDY status, it initiates the message transmission phase. The source DCM transmits the message continuously into the network at the data rate supported by the iPSC/860 channels until the end of the message or until a “not ready” signal is transmitted by the destination DCM. The message is not buffered in any of the intermediate DCM’s.

Figure 7.4: Acknowledgement phase
Releasing the channels

To complete the routing operation, the source DCM appends a checksum word to the message to detect hardware failures. As soon as the checksum word is transmitted, the source DCM releases the outgoing channel reserved for that message. The status path is also released at the same time. Each intermediate DCM in the path also releases the channels reserved for the message on the fly, after forwarding the checksum word.

The destination DCM receives the the checksum word, verifies the message and forwards it to the local processing node.

7.2.3 Message protocol for short messages

In the iPSC/860, all application and system messages of 100 bytes or less use a one-trip protocol [30], which consists of the four-phase protocol described in Section 7.2.2. The operating system on each node maintains short message buffers, and allocates some of these buffers to receive messages from every other node in the system.

When a node wants to send a short message to another node and a short message buffer is reserved for it, the source node immediately initiates the message transfer using the four-phase one-trip protocol. Every node periodically returns the number of short message buffers reserved for a source node.

7.2.4 Message protocol for long messages

Messages that are longer than 100 bytes are not guaranteed space in the destination DCM. Therefore, long messages use a three-trip protocol. Each of the three trips is identical to the protocol described in Section 7.2.2 and used for short messages.

When a long message has to be transmitted, the system sends out a control message to the destination. This is the first trip of the protocol. The control message
consists of a 28-byte software header including the hardware routing probe, and serves as a proxy for the entire message. The four-phase protocol transfers the control message to the destination node. The control message is saved in a short message buffer, and it requests space for the entire message.

If a receive is posted for the message, or if enough memory space is available, the system on the destination node sends back a control message to the source node requesting the rest of the message. This initiates the second trip of the protocol. This control message also consists of a 28-byte software header, and is transferred by using the four-phase protocol. If the source and destination nodes are not nearest-neighbor nodes, the second trip uses a different path in accordance with the e-cube routing algorithm.

When the source node receives the control message from the destination, it initiates the third trip of the protocol, and transfer the entire data.

Thus, in the long message protocol, the channels are arbitrated independently for each trip, and are reserved only for the duration of that trip.

7.3 NETSIM model for the iPSC/860 network

A simulation model for the iPSC/860 has been developed using the primitives provided by NETSIM.

7.3.1 Model for the switch and the channel

Since a full crossbar switch is used in each DCM, a crossbar switch is built using multiplexer and demultiplexer modules provided by NETSIM. A blocked routing probe is buffered in an intermediate DCM as described in Section 7.2.2. Therefore, buffers
capable of holding the routing probe are provided in the NETSIM model. Figure 7.5 shows the NETSIM switch architecture for a 3-dimensional cube.

![Figure 7.5: NETSIM switch model](image)

7.3.2 Simulation algorithm

The wormhole routing simulation algorithm described in Section 4.1 is modified to take into account the switching scheme used in the iPSC/860.

The entire message is simulated using the dynamic implementation described in Section 4.1.2. Two simulation processes are created for each message, one for the head of the message and one for the tail of the message.

The process associated with the head of the message simulates the first two phases of the routing operation described in Section 7.2.2, establishing the message and the status paths. It moves the routing probe from the source node to the destination node. However, unlike the algorithm described in Section 4.1.2, the process associated with
the head of the message does not schedule the process associated with the tail until
the routing probe reaches the destination node.

The process associated with the tail of the message is scheduled after the mes-

gage and status paths are established. This process simulates the third and fourth
phases of the routing operation described in Section 7.2.2, transferring the data to
the destination node and releasing the message and status paths.

7.3.3 Simulation model for short messages

To simulate short messages on the iPSC/860, a single packet is created and transferred
across the network model. The process associated with the head of the message
transfers a header, corresponding to the 32-bit routing probe. The process associated
with the tail of the message transfers the number of flits corresponding to the size of
the message.

7.3.4 Simulation model for long messages

On the iPSC/860, messages longer than 100 bytes use the three-trip protocol described
in Section 7.2.4. The NETSIM model for long messages simulates the three-trip
protocol.

When a message longer than 100 bytes is sent by a user process, the model creates
a 28-byte packet to simulate the first control message. The two processes associated
with this packet transfer the 4-byte routing probe and the 24-byte software header
from the source to the destination.

The process on the destination node receives this control packet, and initiates the
second trip of the protocol by sending another 28-byte packet to the source node. The
two processes associated with the second packet transfer the 4-byte routing probe and
the 24-byte software header from the destination to the source.
When the source receives the second packet, it initiates the third trip of the protocol. It creates a packet the size of the message. The process associated with the head of the packet transfers the 4-byte routing probe while the process associated with the tail of the packet transfers the number of flits corresponding to the size of the message.

7.3.5 Network delays for the simulation model

The NETSIM simulation model requires network delay parameters such as the flit-transfer and the arbitration delays. The flit transfer delay is the time taken to transfer a flit from one switch to the other. The arbitration delay is the time taken to arbitrate between several requests at a multiplexer module.

A simple communication benchmark program called the echo test [10] [15] is used to measure the communication rates of the iPSC/860 system. A test node on the iPSC system sends a message to an echo node. The echo node receives the message and sends it back to the test node. The test node measures the time interval between the send and the receive. This experiment is repeated 10000 times and the average elapsed time is computed. One half of this elapsed time gives the time taken by the message to go from the test node to the echo node. The iPSC/860 message passing routines, csend() and crecv() are used to send and receive the messages. The time is measured by the dclock() routine.

The echo test was run on a 32-node iPSC/860 machine for various message sizes and for pairs of nodes that are adjacent, one hop, two hops or three hops apart. The experimental data obtained from the hypercube is fitted by a least-squares fit to a straight line described by the following performance parameters. Assume the time taken to transfer a message of length $n$ bytes is $t$ seconds. Then, the dependence of
$t$ on $n$ can be described by the linear equation

$$t = t_0 + n/r_\infty$$  \hspace{1cm} (7.1)

where $t_0$ is the startup latency measured in seconds and $r_\infty$ is the asymptotic bandwidth measured in Mbytes/sec.

Data for short messages

For messages less than 100 bytes the echo test is repeated for adjacent nodes and for nodes that one, two and three hops away. This data is plotted in Figure 7.6. For

![Graph showing transmission time vs. message size](image)

**Figure 7.6: Transmission time for messages less than 100 bytes**

messages between adjacent nodes, the experimental data is fitted by Equation 7.1 with the following parameters.

$$t_0 = 76 \mu s, \quad r_\infty = 2.3 \text{ MB/second}$$  \hspace{1cm} (7.2)

For multi-hop messages the data is fitted by Equation 7.1 with the following parameters.
• Single hop

\[ t_0 = 86 \mu s, \ r_\infty = 2.3 \text{ MB/second} \]  \hspace{1cm} (7.3)

• Two hops

\[ t_0 = 96 \mu s, \ r_\infty = 2.3 \text{ MB/second} \]  \hspace{1cm} (7.4)

• Three hops

\[ t_0 = 106 \mu s, \ r_\infty = 2.3 \text{ MB/second} \]  \hspace{1cm} (7.5)

From Equations 7.2 - 7.5, it is clear that the messages have a 10 \( \mu \text{s} \) overhead for every additional hop. This is the time required to perform the routing, to arbitrate between conflicting requests and to transfer the routing probe from the input of the switch to the output of the switch. Assuming that a flit in the model represents one byte, a data rate of 2.3 MB/second implies that the time to transfer a flit from one switch to another is 0.41 \( \mu \text{s} \). Since the routing probe is 4 bytes, the time to transfer the routing probe through a switch is 1.64 \( \mu \text{s} \) while the routing delay and the arbitration delay through the switch model is 8.36 \( \mu \text{s} \). Since NETSIM requires the arbitration and routing delays be integer multiples of the flit transfer delay, this 8.36 \( \mu \text{s} \) delay translates to a routing and arbitration delay of 20 flit-transfer cycles.

Data for long messages

Similar data is collected for messages longer than 100 bytes and the data is plotted in Figure 7.7.

Since messages longer than 100 bytes use the three-trip protocol the startup latency for the messages increases. For messages between adjacent nodes, the data in Figure 7.7 is fitted by Equation 7.1 with the following parameters.

\[ t_0 = 212 \mu s, \ r_\infty = 2.7 \text{ MB/second} \]  \hspace{1cm} (7.6)
Figure 7.7: Transmission time for messages longer than 100 bytes

For multi-hop messages the data is fitted by Equation 7.1 with the following parameters.

- Single hop
  \[ t_0 = 237 \mu s, \ r_\infty = 2.7 \text{ MB/second} \] (7.7)

- Two hops
  \[ t_0 = 267 \mu s, \ r_\infty = 2.7 \text{ MB/second} \] (7.8)

- Three hops
  \[ t_0 = 297 \mu s, \ r_\infty = 2.7 \text{ MB/second} \] (7.9)

Thus, long messages have a 30 \( \mu s \) overhead for every additional hop. Since the long message uses the three-trip protocol, it incurs the 10 \( \mu s \) overhead for routing, arbitration and transfer of the routing probe on each of the three trips, giving a total of 30 \( \mu s \) overhead. Longer messages have a higher data rate of 2.7 MB/second that gives a flit transfer delay of 0.357 \( \mu s/\text{flit} \).
Each trip of the long message protocol incurs a 10 $\mu$s delay for routing, arbitration and transfer of the routing probe. This gives the same values of 1.64 $\mu$s for transferring the routing probe across the switch, and 8.36 $\mu$s (20 flit-delay cycles) for routing and arbitration.

In the long message protocol, the first two trips transfer control messages that are 28 bytes long. Therefore, the model for longer messages uses the 2.3 MB/sec data rate for the first two trips. The third trip actually transfers the message and the model uses the 2.7 MB/sec data rate to transfer this data.

Software Overhead in message passing

The software overhead associated with each message transmission operation includes the time to generate the message header, set up the DMA controller for the message, check the buffer pointers in the send and the receive calls and copy the data. On the iPSC/860 system, it is not possible to determine the delays involved in each of these system operations. Therefore, in order to model the software overhead, it is assumed that the software overhead is the time remaining after accounting for all the hardware delays in the network.

Consider the example of an 8 Kbyte message transmitted between nodes 0 and 3 in a contention-free network. This message uses the three-trip protocol and the transmission time for the message, determined by the echo test, is 3184.24 $\mu$s. The various hardware delays are computed as follows. Each trip incurs a 10 $\mu$s routing and arbitration delay at the source DCM, the intermediate DCM and the destination DCM, giving a total of 90 $\mu$s. In the first two trips, a 24-byte control message is transferred at 2.3 MB/second, accounting for a total delay of 19.68 $\mu$s. In the third trip, the 8K message transferred at 2.7 MB/second takes 2927 $\mu$s. Thus, a total of
3036.68 $\mu$s is accounted for by the hardware delays. The unaccounted delay of 147.59 $\mu$s is assumed to be the software overhead.

The software overhead is independent of the number of hops the messages makes to get to the destination. The software overhead for various message sizes is computed as described above and the delay value is used in the model. The table in Figure 7.8 shows the software overhead for various message sizes.

<table>
<thead>
<tr>
<th>Message size in bytes</th>
<th>Software overhead in $\mu$s</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>98.5136</td>
</tr>
<tr>
<td>128</td>
<td>96.0963</td>
</tr>
<tr>
<td>256</td>
<td>100.808</td>
</tr>
<tr>
<td>512</td>
<td>110.003</td>
</tr>
<tr>
<td>1024</td>
<td>128.204</td>
</tr>
<tr>
<td>2048</td>
<td>155.648</td>
</tr>
<tr>
<td>4096</td>
<td>148.269</td>
</tr>
<tr>
<td>8192</td>
<td>146.915</td>
</tr>
<tr>
<td>16384</td>
<td>146.217</td>
</tr>
<tr>
<td>32768</td>
<td>145.014</td>
</tr>
</tbody>
</table>

Figure 7.8: Software overheads for various message sizes

7.4 Results

The network parameters computed in Section 7.3.5 were built into the NETSIM model, and several experiments were run on the iPSC/860 system and repeated on the NETSIM model. The message latencies predicted by the model for various message sizes and traffic conditions were compared to the timings from the iPSC/860 system to validate the model.
7.4.1 Traffic pattern with no contention

The NETSIM model was first validated against a traffic pattern with no contention for any of the network resources in order to establish the base accuracy of the model.

The echo test described in Section 7.3.5 generates a traffic pattern that has no contention and was repeated on the NETSIM model for various messages sizes. The test and echo nodes were selected such that the path length between the source and the destination nodes is varied. The message latencies predicted by the NETSIM model are compared to the actual message latencies measured on the iPSC/860.

Figure 7.9 shows the performance of the NETSIM model for messages less than 100 bytes, the short messages that use the one-hop protocol. The X-axis represents

![Graph showing error in latency prediction for short messages](image)

Figure 7.9: Error in the latency prediction for short messages

the size of the message in bytes. The Y-axis represents the relative error between the latency value predicted by the NETSIM model and the latency measured on the iPSC/860 network. The relative error is expressed as a percentage, and the message latencies predicted by the model are within 0.6% of the latencies measured on the hypercube.
Figure 7.10 shows the performance of the NETSIM model for long messages that use the three-hop protocol. The X-axis represents the size of the message in bytes and the Y-axis is the relative error in the latency between the NETSIM model and the iPSC/860 network. The message latencies predicted by the model are within 0.7% of the latencies measured on the hypercube.

**Figure 7.10: Error in the latency prediction for long messages**

7.4.2 Traffic pattern with contention for one link

The NETSIM model is validated against a simple contention model in which messages contend for one link in the network. An experiment with single link contention is set up as shown in Figure 7.11. Node 0 sends a message to node 3 which passes through the router attached to node 1. Node 1 sends a message to node 7 which passes through the router attached to node 3. The two messages contend for the channel between the routers attached to nodes 1 and 3. The experiment is repeated such that each node sends 10,000 messages. The node starts sending the next message as soon as the previous message completes the round trip.
Figure 7.11: Traffic pattern with contention for one link

Figure 7.12 shows the performance of the model for messages less than 100 bytes. The X-axis represents the message size in bytes and the Y-axis represents the relative error in the latency between the NETSIM model and the iPSC/860 network for the traffic pattern shown in Figure 7.11. The model is off in its prediction by about 2%.

Figure 7.12: Error in the latency prediction for short messages with one-link contention traffic pattern

Figure 7.13 shows the performance of the model for the single link contention traffic with messages longer than 100 bytes. The model is off in its prediction of the
message latency by about 6%.

7.4.3 Bit-complement traffic pattern

This section validates the NETSIM model against the bit-complement traffic pattern. In this traffic pattern, a source node determines its destination node by complementing all the bits in its binary address. The bit-complement permutation for node 6 is node 1, since the binary representation of 6 is 110, which when complemented gives 001, the binary representation of node 1.

The experiments with bit-complement permutation traffic pattern are run on an 8-node cube. The table in Figure 7.14 shows the destination for each source node in the 8-node hypercube. Since the source and the destination addresses differ in all the bits, the message is routed in all the dimensions of the cube.

The message latency on the iPSC/860 is measured by timing the echo message between the source and the destination nodes. The experiment is repeated such that each node sends out 10,000 messages to the designated destination node. A
<table>
<thead>
<tr>
<th>Source Node (Address)</th>
<th>Destination Node (Address)</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 (000)</td>
<td>7 (111)</td>
</tr>
<tr>
<td>1 (001)</td>
<td>6 (001)</td>
</tr>
<tr>
<td>2 (010)</td>
<td>5 (101)</td>
</tr>
<tr>
<td>3 (011)</td>
<td>4 (100)</td>
</tr>
<tr>
<td>4 (100)</td>
<td>3 (011)</td>
</tr>
<tr>
<td>5 (101)</td>
<td>2 (010)</td>
</tr>
<tr>
<td>6 (110)</td>
<td>1 (001)</td>
</tr>
<tr>
<td>7 (111)</td>
<td>0 (000)</td>
</tr>
</tbody>
</table>

Figure 7.14: Bit-complemented destination addresses for an 8-node hypercube

source node initiates the next message as soon as the previous message completes its round trip. Thus, the network is driven to saturation. Figures 7.15 and 7.16 show the performance of the model for the saturated bit-complement traffic pattern. The

![Graph showing % error for message size in bytes](image)

Figure 7.15: Error in the latency prediction for saturated bit-complement traffic for short messages

X-axis represents message size in bytes and the Y-axis represents the relative error in the message latency predicted by the NETSIM model. The model is off in its
Figure 7.16: Error in the latency prediction for saturated bit-complement traffic for long messages

prediction of message latencies by about 1% for short messages and by about 4% for long messages.

Figures 7.17 and 7.18 show the performance of the NETSIM model for a lightly loaded network with the bit-complement traffic pattern. Each node waits long enough

Figure 7.17: Error in the latency prediction for lightly loaded bit-complement traffic for short messages

for all the messages to drain out of the network before initiating the next message.
The message latencies estimated by the model for the lightly loaded network are closer to the latencies measured on the iPSC/860 compared to the estimates for the saturated network.

![Graph showing error in latency prediction for lightly loaded bit-complement traffic for long messages](image)

**Figure 7.18**: Error in the latency prediction for lightly loaded bit-complement traffic for long messages

### 7.4.4 Matrix transpose traffic pattern

The matrix transpose traffic pattern [4] is also used to validate the NETSIM model. This traffic pattern is generated when nodes exchange data to compute the transpose of a matrix. A 16-node hypercube is used to run the validation experiments. The elements of a 4x4 matrix are assigned to the nodes. All the nodes with elements above the diagonal exchange data with the nodes below the diagonal. The nodes with the diagonal elements do not send any messages. The traffic pattern on the 16-node hypercube is shown in Figure 7.19.

The diagonal nodes 0, 5, 10 and 15 do not send any messages. Nodes above the diagonal send messages to nodes below the diagonal. For example, node 1 sends and receives messages from node 4. The other pairs that communicate are nodes 2 and 8,
Figure 7.19: Matrix transpose traffic pattern for a 16-node hypercube

nodes 3 and 12, nodes 6 and 9, nodes 7 and 13 and nodes 11 and 14. The message from node 1 to 4 goes through the router on node 0. The message from node 3 to 12 goes through the routers on nodes 2, 0 and 4. Thus, these two messages contend for the channel between nodes 0 and 4. For long messages, the second trip of the three-trip protocol uses a different path to get to the source node from the destination node increasing the contention for the network channels.

The message latency on the iPSC/860 is measured by timing the echo message between the source and the destination nodes. The experiment is repeated such that each node sends out 10,000 messages. The source node waits for the messages to drain out of the network before initiating the next message. Figures 7.20 and 7.21 show the performance of the NETSIM model for the lightly loaded network with matrix transpose traffic pattern.

7.4.5 Summary of the results

The NETSIM model for the iPSC/860 interconnection network estimates the message latencies in a contention free network with an error of 0.6% or less for message sizes up to 32 Kbytes. The error in the predicted latencies increases with increase in
contention in the network. In a network with contention, message headers get blocked in intermediate routers. Since information about how a blocked header is handled inside a router is not available, the delays associated with the blocked header are not modeled accurately in the NETSIM model. Therefore, for a bit-complement traffic pattern the model is off in its prediction by about 4% and for a matrix transpose traffic pattern the model is off in its prediction by about 7%.
Chapter 8

Conclusion

A modular, general-purpose interconnection network simulator called NETSIM has been presented in this thesis. NETSIM provides five basic network building blocks that can be interconnected to build customized switches. These switches can be used to build networks of arbitrary topology and size. A NETSIM model can be simulated in a stand-alone mode, or it can be used as an integral part of PARCSIM in an execution-driven simulation environment.

Section 8.1 summarizes the results of this thesis and Section 8.2 suggests future extensions to the simulator.

8.1 Summary

This thesis emphasizes efficient simulation of complex switching schemes like wormhole routing. Two implementation techniques have been developed and evaluated in this thesis. The dynamic process implementation technique assigns simulation processes to the packets in the network while the static process implementation assigns simulation processes to the modules in the network. For networks with single-flit buffers, the dynamic implementation is faster than the static implementation by about 15%. For networks with large flit buffers, the dynamic implementation is about 10% faster than the static implementation.

A detailed simulation model for wormhole routing models the asynchronous movement of the individual flits of the packet. As a result, the simulation overhead increases with increases in the packet size, the buffer size and the input load. The overhead incurred in simulating wormhole routing is about 10 times the overhead in-
curred in simulating simple switching schemes like store and forward routing. In order to reduce this overhead, four approximate techniques to simulate wormhole routing are presented, and the tradeoff between accuracy and speedup has been evaluated.

Buffer approximation approximates the movement of the flits inside a buffer, and reduces the process-scheduling overhead. This technique is not applicable to networks with single-flit buffers. For networks with large buffers, the speedups range from 2 to 5 with errors within 1%.

Tail-flit approximation approximates the movement of the tail flit of the packet after the head flit leaves the network, and reduces the bubble-management overhead. For networks with small flit buffers, the simulation overhead is independent of the packet size with errors within 0.8%. For networks with large buffers, the speedups range from 1.5 to 3 with errors within 8%.

The combined approximate model combines buffer approximation and tail-flit approximation into a single model. For networks with small buffers, the combined approximate model is identical to the tail flit approximation model. However, for networks with large flit buffers the speedups range from 3 to 6 with errors within 12%.

The extended approximate model further reduces the simulation overhead in the combined model by eliminating the process-synchronization overhead. For networks with small buffers, the extended approximate model is 40% faster than the combined approximate model. For networks with large flit buffers, the extended approximate model is almost identical to the combined model.

Thus, the choice of the best approximate model for a given network depends on the buffer size, the packet size and the load. For networks with small buffers, the extended approximate model offers the best compromise between speedup and accuracy, irrespective of the packet size and the load. For networks with large buffers
and heavy load, buffer approximation is the best. For networks with large buffers and light load, the extended approximate model is the best for small packets while the combined model is the best for large packets.

The NETSIM validation model for the iPSC/860 interconnection network estimates the message latencies in a contention free network with a 0.6% error. The accuracy of latency estimates for traffic patterns with contention is within 7%.

8.2 Future Work

Multicast [23] communication is frequently used by parallel algorithms executing in a large system to deliver messages from a single source node to arbitrary number of destination nodes. Commercial systems such as the nCUBE-2 support broadcast using wormhole routing. The routing capabilities provided by NETSIM should be modified to support multicast and broadcast communications.

The use of virtual channels in wormhole routed networks increases the throughput of the network [6]. Virtual channels are also used for developing deadlock-free adaptive routing algorithms for wormhole routed networks [26]. The wormhole routing algorithm implemented in NETSIM should be extended to support virtual channels.
Bibliography


